

**NASA CONTRACTOR
REPORT**

NASA CR-1520



NASA CR-1520

0.1

0060700

TECH LIBRARY KAFB, NM

FLIGHT COMPUTER AND LANGUAGE PROCESSOR STUDY

by Raymond J. Rubey, William C. Nielsen, and Laurel Bentley

Prepared by

LOGICON, INCORPORATED

San Pedro, Calif.

for Electronics Research Center

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION • WASHINGTON, D. C. • MARCH 1970

✓ call no.
NASA CR-1520
TECH LIBRARY KAFB, NM



0060700

✓ FLIGHT COMPUTER AND LANGUAGE PROCESSOR STUDY

By Raymond J. Rubey, William C. Nielsen,
and Laurel Bentley

✓ mar 70

Distribution of this report is provided in the interest of
information exchange. Responsibility for the contents
resides in the author or organization that prepared it.

Prepared under Contract No. NAS 12-2005 by
w.e. ✓ LOGICON, INCORPORATED
San Pedro, Calif.

for Electronics Research Center

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

For sale by the Clearinghouse for Federal Scientific and Technical Information
Springfield, Virginia 22151 - Price \$3.00

CONTENTS

(continued)

| | |
|--|-----|
| Assignment | 71 |
| Fixed-Point Assignment Using TEMP | 75 |
| Scaling Operator | 76 |
| Multiple Assignment | 79 |
| Nonscalar Assignment | 81 |
| Exchange Assignment | 85 |
| Program Control | 87 |
| Statement Labels | 88 |
| GOTO Statements | 89 |
| Switched GOTO Statements | 90 |
| Conditional Statements | 92 |
| Loop Statements | 95 |
| Inhibit/Enable Statements | 98 |
| Chronic Statements | 100 |
| STOP Statements | 102 |
| Subprograms | 103 |
| Procedure Declarations | 104 |
| Procedure Calls | 107 |
| Functions | 110 |
| Close Declarations | 111 |
| Close Calls | 113 |
| Library Subprograms | 114 |
| Directives | 119 |
| Debugging Directives | 120 |
| Timing Directives | 121 |
| Optimization Directives | 122 |
| Direct Code Directives | 123 |
| Part III - Compiler Considerations | 125 |
| Optimization Techniques | 125 |
| Compiler Outputs | 129 |
| Compiler Capacities | 130 |
| Implementation-Dependent Language Features | 131 |
| Appendix A - Index for Part II | 133 |

FLIGHT COMPUTER AND LANGUAGE PROCESSOR STUDY

By Raymond J. Rubey, William C. Nielsen,
and Laurel Bentley

Logicon, Inc.
San Pedro, California

PART I - SUMMARY

This report describes the results of a study to define a higher level programming language suitable for the development of real-time aerospace programs. The programmer's reference manual for such a language and compiler considerations for the language's implementation are presented in the succeeding parts of this report.

Having being intended to be of immediate, practical benefit, the study was oriented toward computers and applications of the present and near future. While not considered a special-purpose machine, the typical aerospace computer has a relatively small memory and a fairly limited instruction set, with no hardware floating-point operations and minimal instructions for logical decision functions and internal index register manipulation. With regard to the problems to be solved, these consist principally of arithmetic calculations and logical decisions, with significantly fewer text manipulation and table look-up operations. Guidance and navigation functions are performed by repeating the appropriate calculations at a relatively slow major cycle frequency; other functions such as input/output processing and control are performed at much higher minor cycle frequencies. The required response times between input and output, which are functions of the overall vehicle system design, are generally very short -- on the order of a few seconds at most.

The study plan itself consisted of using two candidate languages to code portions of a representative problem designed to be solved on a current computer; analyzing the resultant programs to select a base language for further definition; using the language thus defined to reprogram the same problem; and analyzing the second set of programs to enable further language definition and refinement. Thus the first step to be taken was to choose the candidate languages. SPL, which had been defined for the application area, was selected, as was PL/I, the latter because it included many of the real-time capabilities found in SPL. Other possible candidates were eliminated, FORTRAN, for example, because most of its functions can be accom-

plished using the richer PL/I, and JOVIAL because the developers of SPL had already indicated that it was inadequate and had found it necessary to make many extensions to basic JOVIAL in the definition of SPL. Then, to serve as the benchmark problem, portions of the Titan IIIC programming specification developed by The Aerospace Corporation were selected. These consisted of a set of typical guidance equations, program blocks for engine output command calculations and staging sequencing, and a portion of the main control logic flow diagram, this last to enable determining the languages' suitability for coding executive functions. The overall 1-second major cycle logic was specified, along with the executive program to control proper phasing of the major cycle and 5-, 10-, and 20-millisecond minor cycle functions. As is typical of aerospace programming specifications, also included was complete information specifying the range and accuracy required for all program parameters, including where extended precision was to be used, and the critical timing requirements to be met.

Selecting the Base Language

Four programmers independently coded the representative problem, two in each language. They were instructed to spend as much time as possible on its more difficult and aerospace-peculiar aspects, avoiding repetitive operations that would not lead to meaningful conclusions or that could be handled in some way not provided for in the assigned language. None of the programmers was able to code the entire benchmark problem in the two weeks allotted, but each succeeded in coding substantially more than would have been possible using assembly language. Thus both SPL and PL/I were shown to offer significant benefits, chiefly through their relatively simple assignment and control statements. Nevertheless, both languages had serious deficiencies which prevented complete, direct solution of the problem. It was clear that the tricks and circumlocutions necessary to overcome these deficiencies would negate the hoped-for benefits of using a higher level language.

While the programmers were able to specify the equations easily, describing the data attributes necessary to control precision throughout the steps of a calculation presented difficult problems. This reflects the fact that both languages are oriented to having the bulk of the calculations done in floating rather than fixed point. Many of the fixed-point difficulties were similar in that the relatively simple way of doing an operation in assembly language was not available in either SPL or PL/I; to overcome these problems, the programmer would have to break the program into steps almost as numerous and as detailed as he would when coding in assembly language. For example, one operation the programmers needed to be able to do but could not, at least not without a great deal of difficulty,

was to define a fixed-point variable with the binary point outside the number of bits actually allocated to the item. This may be desirable for variables having either very small or very large values; as a hypothetical example of the latter, if the coordinates of a vehicle's position are measured in feet and it is sufficient to maintain each to a precision of only 16 feet, the programmer would wish to scale the coordinates so that the binary point is four places to the right of the computer word's least significant bit in order to allow the greatest range of values for position.

Another problem arose when it was necessary to give a variable one scaling for a particular mission phase and another scaling for a subsequent phase. For example, it may be necessary to maintain a vehicle's position to one precision during near-earth maneuvers and to a much less accurate precision subsequently. However, neither PL/I nor SPL offered a means of dynamically rescaling variables without a great deal of coding duplication.

A third problem came about with the use of double-precision accumulation of products. To exemplify the mechanism involved, the multiply operation in the typical aerospace computer will generate a double length for the intermediate products of the following equation:

$$(V)^2 = (V_x)(V_x) + (V_y)(V_y) + (V_z)(V_z)$$

Making use of the double-precision addition command available in many computers enables accumulating the sum in double precision to obtain improved accuracy; however, this requires one more instruction per term and causes some increase in execution time. In some cases it is not desirable to pay this penalty to achieve improved accuracy; in other cases using double precision is mandatory, such as in matrix operations when the intermediate products are of opposite sign but nearly equal in magnitude. The programmers required control over the choice - - something that neither PL/I nor SPL offered simply and directly.

A fourth problem area concerned the temporary saving of intermediate results in common locations. Neither language provided a convenient mechanism for doing this with fixed-point quantities.

These, then, have been a few of the problems encountered with fixed-point variables in coding the test problem. Turning now to constants, a peculiarity of aerospace programming is found in the existence of two distinct types of constants: absolute and modifiable. Absolute constants, such as the coefficients in a polynomial approximation for the sine of an angle, are those which are very unlikely to be altered when the program itself is changed to

accommodate it to a new mission. Modifiable constants, for example, accelerometer nonlinearity compensation terms, are those which are expected to be different for different missions. The program is written and checked out using nominal values for such mission-dependent constants, and the actual values are loaded directly into memory when they become known, often shortly before the mission. To permit this to be done, the programmer must be able to specify the attributes of such constants (required precision and maximum value as well as the nominal numeric value) as completely as he can those of variables. That is, he needs to be able to specify enough information so that modifiable constants could be treated similarly to variables in the automatic scaling algorithms of a compiler; if he could not do so, the program would have to be recompiled every time the value of a single mission-dependent constant was changed. The alternative provided in PL/I and SPL -- defining such constants as variables having preset values -- was found undesirable because it would hinder any compiler optimization functions. Also pointed up by an analysis of the test problem programming was the need for an ability to define the value of a constant as a function of other constants; this would simplify programming in many cases and, by enabling automatic change of such dependent constants, would help to reduce errors introduced when constants are modified.

Other language deficiencies were found in many areas. While the aerospace programmer is concerned chiefly with incremental inputs, telemetry, and discrete output commands, the input/output capabilities of both languages were not easily related to the requirements of the benchmark problem, instead emphasizing files, records, and peripheral devices. Similarly, review of the limited debugging capabilities showed that they were not oriented to aerospace programs, for which much of the debugging is done using computer simulators rather than the actual computers. Also, neither language provided sufficient programmer control over object code optimization; however, PL/I contained a built-in possibility for extending its existing capabilities in this area.

Both languages were discovered to have numerous features that were not required for coding the representative benchmark problem. That there should be many features in PL/I that were not particularly useful for the application area was expected, but SPL also had a great many features of little or no utility, among them complex array declarations, table declarations, alternative forms of many statement types, text manipulation, and automatic and controlled storage. Both provided many needed capabilities, and both were deficient in the areas of fixed-point data declaration and arithmetic control. PL/I was superior as regards optimization control but lacked desirable features of SPL, such as built-in matrix operations, decision tables, and a simple method of interfacing with direct code.

Overall, SPL was found to have more features that might be useful and fewer that were clearly unsuitable; hence it was selected as the base for development into a concise aerospace programming language. The choice was influenced by the fact that the Air Force and System Development Corporation were proceeding with the development of SPL, and it was expected that continuing cooperation among the two government agencies and their contractors might result in further modification of SPL to make it more suitable and at the same time compatible with the corresponding NASA language. The NASA language developed on the basis of this work and the subsequent study phases was given a distinct name, CLASP (Computer Language for Aeronautics and Space Programming), to minimize confusion between it and the continually evolving SPL. Many modifications to SPL have been initiated by System Development Corporation as a result of the work discussed here, and most of the deficiencies discussed above no longer exist in SPL. The objective of having CLASP be a proper subset of SPL has been achieved in large part. In the absence of a standardization control authority, however, the compatibility of various compiler implementations will almost certainly vary, particularly with regard to semantic differences.

CLASP's Capabilities

While CLASP's basic structure is similar to that of other higher order languages -- the assignment and logical control statements, for example, would not be surprising to a FORTRAN programmer -- it provides many features that are either unique in themselves or are used in unique ways. Only these unique capabilities will be discussed here.

CLASP allows the programmer to declare the attributes of fixed-point data items such that the code generated by a compiler will perform the indicated arithmetic with the required accuracy and without excessive penalties in object code size or execution time. The CLASP programmer can specify the minimum total number of bits, m , and the minimum number of fractional bits, f , to be allocated to each fixed-point item. In practice, he should specify for each data item only the minimum number of bits to be allocated for the expected range of values and the necessary accuracy. While the total bits, m , needed to allow data storage at the required precision might not be a multiple of the number of bits per word, it would require fewer instructions and less execution time to allocate storage for the item in increments of full words. It is envisioned that a CLASP compiler will not use any excess bits to allocate more than one data item to a single computer word unless explicitly directed to do so, but instead will use them to generate a consistent set of scalings that minimizes intermediate shifting.

As an example, a minimum intermediate scaling readjustment would be required in evaluating the expression $A + B - C$ if all three variables had

the same number of integer and fractional bits, even though the programmer had not declared them so. The minimum number of integer bits that must be allocated for each is implicit in the programmer's declaration, and is the total number of bits declared, m , less the number of fractional bits declared, f , and less the sign bit. For a computer with a 24-bit word size, it can be seen from Table I that there would be three ways of storing A , B , and C if each had been declared to be 22 bits in size. It can also be seen that there is but one possible scaling that is common to all three variables: 11 integer bits and 12 fractional bits. For simple cases such as this, a compiler can try all possible scalings and select the common scaling, within the range and precision requirements specified by the programmer in the data declaration, that permits the minimum intermediate scaling readjustment. The resulting allocations will not be the exact allocations he has specified, but will be both consistent and in excess of his declared requirements. For more complex expressions containing variables that also appear in other expressions, a compiler will require elaborate heuristics and algorithms to select the scalings. The programmer will have to remember that those selected by the compiler are guaranteed to be within his specified ranges, with the compiler choosing the best scalings when, by declaring the number of bits to be smaller than some multiple of the computer's word size, he has given it the flexibility to do so.

As mentioned in the discussion of the base language selection, another fixed-point problem existed with regard to the use of registers or data words for temporary storage. Normally, when a fixed-point assignment of the form $\alpha = \beta$ is made, before storage takes place the computed value of β must be adjusted by shifting in order that the binary points of α and β will be aligned. Such a readjustment should not be made, however, if α is a temporary variable that might be used in many places in the program and with different attributes desired for each place. The solution provided for this problem in CLASP is to declare such temporary variables as data items having the attribute TEMP. Doing this has the result that such a variable will assume the temporary attributes of the expression to the right of the equals sign until such time as a new temporary assignment is made to that variable.

To solve the problems relating to modifiable constants, CLASP allows them to be specified as parameters; absolute constants are specified simply as constants. Parameters may be changed before program execution without requiring recompilation, while constants are fixed at compile time. Both are likely to be assigned to read-only storage if the aerospace computer has such a structure.

The fact that a constant's value does not change without recompilation means that a CLASP compiler will be able to determine the permitted range of scalings solely from the value given. If, for example, a constant's value

TABLE I
POSSIBLE ALLOCATIONS FOR VARIABLES IN
A COMPUTER WITH A 24-BIT WORD SIZE

| Variable | Data Declaration | | | Possible Scalings (i:f) | | | | |
|----------|----------------------|---------------------------|------------------------------|-------------------------|-------|-------|-------|-------|
| | Total Bits (m) | Fractional Bits (f) | Integer Bits (i=m-f-1) | | | | | |
| | | | | 9:14 | 10:13 | 11:12 | 12:11 | 13:10 |
| A | 22 | 10 | 11 | No | No | Yes | Yes | Yes |
| B | 22 | 11 | 10 | No | Yes | Yes | Yes | No |
| C | 22 | 12 | 9 | Yes | Yes | Yes | No | No |

is established as 2.5, only two bits would need to be allocated for the integer part and one for the fractional part, greatly increasing the flexibility available to the scaling heuristics and algorithms in finding an optimum set of scalings. Constants might not even appear explicitly in the program; for example, a multiplication by a constant might be replaced by a shift. Parameters, on the other hand, must appear and must be declared with attributes such as the range of values and the precision required, just as variables are declared.

In cases where the scalings derived through the use of the scaling algorithms may be undesirable, CLASP provides a scaling operator to make it possible for the programmer to specify the total number of bits and the number of fractional bits for any intermediate result, just as he can for declared items. Thus in the expression

$$A + (B - C) .S(10, 9) + D$$

the scaling operator `.S(10, 9)` specifies that the size of the intermediate result $(B - C)$ is 10 bits, nine of which must be fractional bits. This capability has been provided for occasions when the programmer has information about intermediate results -- such as the fact that B and C are always about the same size -- which is not supplied in the data declaration but which may be required to generate code that produces results of the required precision. An abbreviated form of the scaling operator can also be used when the programmer wishes to accumulate products in double precision and assign the result to a single-precision variable.

Object code efficiency is of great interest because of the small memory size of the typical aerospace computer and the strict real-time requirements to be achieved by the typical program. Accordingly, many features are incorporated to enable the generation of such efficient object code by a CLASP compiler. Primary among these are three optimization directives that can be applied to any desired area of code: `OPTIMIZE SPACE (n)`, `OPTIMIZE TIME(n)`, and `SIC`. The last of these is provided for indicating that no optimization is to be attempted by the compiler; it is included for use primarily in early program development stages when the programmer is interested in getting a rough idea about program correctness. By placing a `SIC` at the beginning of a program area, the programmer can direct that all other optimization directives within that area are to be ignored.

For the space and time optimization directives, the parameter n serves to specify the degree or level of optimization. Recall that most aerospace programs have functions which must be performed at a high frequency and others which are performed at lower frequencies. Clearly, it is very important to optimize the execution time of the higher frequency functions, and proportionately less

important as the frequency becomes lower. For example, if a control function is to be performed 20 times each second and a guidance function but once each second, the programmer could specify OPTIMIZE TIME (20) for the former function and OPTIMIZE TIME (1) for the latter. In the event that program areas included in the higher frequency functions are to be executed only under special conditions, the programmer can assign to them a relatively low degree of time optimization. With regard to space optimization, this is most likely to be specified for compiling the lower frequency functions.

Among the CLASP features that also have a marked effect on the degree of optimization obtained is the nonscalar subscript, (*), by means of which all elements of a row, column, or plane of an array can be successively referenced. Consider the following set of equations:

$$\begin{aligned} P_x &= k_{11} V_x + k_{12} V'_x + k_{13} V''_x \\ P_y &= k_{21} V_y + k_{22} V'_y + k_{23} V''_y \\ P_z &= k_{31} V_z + k_{32} V'_z + k_{33} V''_z \end{aligned}$$

If variables V_x, V_y, \dots, V_z and the constants $k_{11}, k_{12}, \dots, k_{33}$ are declared as elements of arrays, the single CLASP statement

$$P(*) = K(*, 1) * V(*) + K(*, 2) * VP(*) + K(*, 3) * VPP(*)$$

will accomplish the computations for all three equations. This CLASP statement could be translated by a compiler in two ways. First, it could be replaced by an equivalent statement with normal, single-valued subscripts, and this statement preceded by a loop control statement that would cause it to be executed three times, with the subscript value, initially 0, incremented each time. This would result in compiled code of size s_1 and execution time t_1 . Alternatively, three statements could be generated from the single statement written; these statements would be similar to the given equations (with no subscripting). This alternative would result in compiled code of size s_2 and execution time t_2 . If S were the level of space optimization and T the level of time optimization specified in the appropriate optimization directive statements, the best choice would be the first method if $S \times s_1 + T \times t_1$ were less than $S \times s_2 + T \times t_2$ and the second method if not.

The optimization directives are also used in the generation of fixed-point code. In the absence of any other information (e.g., from the scaling operator), the intermediate rescalings that may be required during arithmetic expression evaluation to resolve otherwise conflicting scalings should be chosen such as to minimize the function

$$T_r = \sum_{i=1}^n T_i r_i$$

where

n = total number of rescaling operations

T_i = time optimization parameter n in effect
for the i th rescaling operation

r_i = execution time for the i th rescaling operation

This function states that scaling readjustments should be done in the region where the degree of time optimization specified is the least. A similar function could be written for the effect of rescaling on size optimization. In a practical case, it will not be necessary to evaluate all possible scaling readjustments to determine the minimum T_r because the problem can be partitioned and individual scaling readjustments determined for individual variables or small groups of variables.

CLASP contains features which some might think of as retrograde steps to machine dependency. These features were added to promote efficiency and because an aerospace computer program of necessity has a close interrelationship with its hardware environment. It is possible in CLASP to assign an identifier to a machine register and declare the attributes (e.g., data type, number of integer and fractional bits) of that register when it is referenced by that identifier. For example, the statement

DECLARE HARDWARE INTEGER, INDEX1=2

would assign the identification INDEX1 to hardware address 2 and specify that it contained integer values. In conjunction with these hardware declarations, the programmer has the capability of reserving the use of registers for his own special purposes. The directive LOCK 2 would prevent the compiler from generating code using the indicated hardware register 2 except where the programmer explicitly referenced it by the declared identifier. He would return the use of that register to the compiler by the directive UNLOCK 2.

Several in-line arithmetic functions are provided for doing elementary operations such as absolute value, rounding, and limiting to a specified range. Logical operators are included for performing logical product, logical sum, exclusive OR, and shifting operations. In the event that the programmer cannot accomplish his objectives using these machine-like operations, he can lapse into in-line assembly code without any attendant inefficiencies due

to linkages. The interrupt capability of the computer, utilized for most aerospace program executives, is handled in CLASP by means of the ON statement; this allows the programmer to declare the means by which the interrupt routine is entered and exited. After the entry mechanism has been declared, interrupt processing is handled by means similar to the normal subroutine capability of the language; thus the executive can be considered as a special case of a subroutine. The LOCK and UNLOCK statements used to reserve and restore the use of machine registers can also be used to inhibit or activate interrupts.

Some of the things that CLASP does not contain may seem surprising. Any superfluous features would be likely to make the language harder to learn and use, to make it more costly to implement, and, most important, to result in concomitant losses in the efficiency of the generated object code. Much attention was therefore paid to the specification of a "bare bones" language adequate to do the job efficiently but containing no frills.

CLASP does not have any built-in input/output operations because of the wide differences in the input/output characteristics of aerospace computers, together with their very application-dependent nature. Their absence is justified by experience with other special-purpose higher level languages: regardless of what may have been specified in the language, actual implementations differ widely because of differing application needs. Input/output operations are accomplished in CLASP by lapsing into direct code and making use of hardware declarations.

CLASP does not have the built-in mathematical functions -- sine, arc-tangent, etc. -- common to other languages. Although these are present in the base language, SPL, they were eliminated in defining CLASP because they would introduce unacceptable inefficiencies: to implement them, it would be necessary to prepare either a general subroutine package containing all functions or individual subroutines for each function. The general package would be inefficient if only a few functions were required by a particular application program, and the individual subroutines inefficient if most functions were required. These inefficiencies are further compounded when such subroutines are required with fixed-point input and output parameters. For example, the fixed-point arctangent function satisfactory in one aerospace application program may be unsuitable in another because of differences in the permitted ranges of arguments, accuracy required, and allowable execution time. In CLASP, mathematical functions may be defined by the same means as any other subroutine; the programmer, however, must supply the procedure specifying how the function is to be calculated, including the precision, range of values, etc., for its arguments. In practice, a library of such functions will

be maintained, to be drawn from as required for any specific program, with additions to the library being made as a need for function subroutines with particular properties occurs.

Compared with SPL, CLASP has many other, although less significant simplifications. Such things as status variables, table declarations, qualified named variables, matrix inversion, and notational substitution directives have been deleted. The conditional statements, allowable subscript expressions, and assignment rules have been greatly simplified. Together with the additions discussed above, these simplifications make CLASP a language that can do the job in the aerospace programming area and can be implemented for the computers of today and the near future without great expense.

PART II - PROGRAMMER'S REFERENCE MANUAL

CLASP is divided into five areas: data declaration, formulas and assignment, control, subprogram definition, and directives. Each major area is composed of specific features. Those of the data declaration area are used to define data items, both singly and in aggregations, and to specify their attributes, including any numeric values. The data items thus defined are used to construct formulas; using assignment statements, it is possible to change the value of a data item by assigning to it the value of an evaluated formula. Control statements are used to indicate the sequence of execution for the assignment and other statements. Subprogram definition statements are used to define areas of the program as subprograms, making it possible for the functions they perform to be called for at any place in the program. Directives are used to specify compiler functions independent of the procedures that the program is to perform.

Following general information pertaining to this manual and to CLASP as a whole, each major language area is briefly described, each such description being immediately followed by detailed descriptions of the specific language features that that language area comprises.

Metalinguage used in format descriptions. -- The metalinguage defined in Table II is used in describing the format of CLASP features.

TABLE II
METALANGUAGE USED IN CLASP FORMAT DESCRIPTIONS

| <u>Notational Form</u> | <u>Example</u> | <u>Definition</u> |
|---|--|--|
| Uppercase letters | DECLARE | A CLASP primitive |
| Lowercase letters | identifier | A class of CLASP elements |
| { } ^{n₁} _{n₂} | {, variable} ² ₀ | Repetitive braces allowing from n ₁ to n ₂ repetitions of the syntax they surround; 0 to 1 repetitions are assumed if values are not specified for n ₁ and n ₂ |
| Stacked symbols | FIXED A | One and only one of the alternatives shown |

As an example of the use of this metalanguage, the following format

$$\text{DECLARE } \begin{matrix} \text{INTEGER} \\ I \end{matrix} \left\{ \begin{matrix} \text{CONSTANT} \\ \text{PARAMETER} \end{matrix} \right\} \{=\text{formula}\}$$

$$\left\{ , \text{identifier} \left\{ \begin{matrix} \text{CONSTANT} \\ \text{PARAMETER} \end{matrix} \right\} \{=\text{formula}\} \right\}_1^\infty$$

which describes the syntax of an integer item declaration, states that the declaration must begin with the primitive DECLARE, be followed by either the primitive INTEGER or the letter I, then optionally followed by the primitive CONSTANT or PARAMETER and an optional field with an equals sign followed by a formula. The next fields, starting with the comma, are all enclosed in braces which have the limits 1 to ∞ . This means that there must be at least one occurrence of the enclosed syntax and that there is no intrinsic limit on the maximum number of occurrences (although any particular implementation will be limited by compiler capacities). One of the many possible declarations satisfying the above syntax is the following:

DECLARE I, ABLE, BAKER CONSTANT

In giving format definitions and examples of CLASP code, lines typed indented from a previous line represent continuations of a statement; lines not indented in this way represent new statements.

Program structure.--A CLASP program begins with a START declarator. Immediately following are the data declarations that constitute the program environment. Following the last such data declaration are the imperative statements and directives that operate on the declared data and that represent the main program. Following the last statement of the main program are programmer-defined procedures, each of which may contain data declared locally. Closes may be declared anywhere in the source program. The last statement in the program is the TERM declarator.

Character set.--The CLASP character set is restricted to the letters A through Z, the numerals 0 through 9, the symbols + - * / , () . ' \$ = and a blank character.

Source code and statement formats.--It is assumed that the CLASP source program will be compiled from punched cards; however, other devices can be used if they have an 80-character logical record length.

Source code may begin in any column from 1 to 72 but must not extend past column 72. Column 73 is the continuation field and is not interpreted

by the compiler. Columns 74 through 80 are usually used for card numbering and program identification and are similarly not interpreted by the compiler.

One statement may extend for more than one coding line, and there is no limit to the number of continuations allowed for any one statement. However, identifiers, numerical quantities, operators, and primitives must not be broken between lines. If column 73 is blank, the compiler assumes that the next line of coding is the beginning of a new statement. If it is not blank, the current statement is continued to the next card. For purposes of continuing literal text, column 72 is considered to be followed by column 1 of the continuation line.

More than one statement may be placed per line by using a \$ terminator between statements.

Blanks between elements may be inserted or omitted as desired provided that their insertion or omission does not result in the creation of new and unwanted identifiers, numbers, or primitives. For example, the following may be written with or without blanks:

```
COUNT + 1
2 + 5
SIX / (TEN + 2)
```

while the following must be written with blanks where indicated by ¢:

```
DECLARE¢FIXED, ABLE¢24 ¢10
A¢EQ¢0
```

Identifiers.--Identifiers provide a means of referring to a specific data item, array of data items, statement, procedure, etc., and are defined by the programmer. The format descriptions of each language feature describe how identifiers are defined and used. CLASP identifiers must begin with a letter which may be followed by from 0 to 7 letters or digits.

Primitives.--CLASP primitives cause certain actions to occur or they serve as a guide in the analysis of the syntax. A complete list of primitives is given in Table III. In this table, an asterisk is used to indicate reserved words, that is, primitives that may not be used as programmer-defined identifiers. Primitives not marked with an asterisk may also be used as identifiers.

Comments.--Except in literal text, comments may be inserted anywhere in the source program where a blank is permitted. A comment is delimited

TABLE III
CLASP PRIMITIVES

An asterisk indicates reserved words that may not be used as programmer-defined identifiers.

| | |
|----------|--|
| A | Fixed-point type declarator |
| *ABS | Absolute value function |
| *AND | Boolean AND operator |
| B | Boolean type declarator; binary constant indicator |
| BOOLEAN | Boolean type declarator |
| *BY | Loop statement step size indicator |
| *CLOSE | Close declarator |
| CONSTANT | Data item attribute |
| *COUNT | Start delimiter for timing directive |
| *DECLARE | Data item declarator |
| DIRECT | Start delimiter for direct code |
| *ELSE | Start delimiter for ELSE statement group in conditional statement |
| *END | End delimiter for conditional and loop statement groups and for direct code |
| *ENDALL | End delimiter for all conditional and loop statement groups not yet terminated |
| *EQ | Relational operator: is equal to |
| *EQUIV | Boolean equivalence operator |
| *EXIT | End delimiter for chronic statement, procedures, functions, and closes |
| F | Floating-point type declarator |
| *FALSE | Boolean constant stored internally as 0 |
| FIXED | Fixed-point type declarator |
| FLOATING | Floating-point type declarator |
| *FOR | Start delimiter for loop statement |
| *GOTO | Unconditional transfer statement |
| *GQ | Relational operator: is greater than or equal to |
| *GR | Relational operator: is greater than |
| HARDWARE | Hardware register operand declarator |
| I | Integer type declarator |
| *IF | Start delimiter for conditional statement |
| INDEX | Index register assignment declarator |
| INLINE | Procedure attribute |
| INTEGER | Integer type declarator |
| L | Location constant indicator |
| *LAND | Bitwise logical AND operator |
| *LIM | Limiting function |

TABLE III
CLASP PRIMITIVES
(continued)

An asterisk indicates reserved words that may not be used as programmer-defined identifiers.

| | |
|-----------|---|
| *LOCK | Inhibit directive |
| *LOR | Bitwise logical OR operator |
| *LQ | Relational operator: is less than or equal to |
| *LS | Relational operator: is less than |
| *LSH | Logical operator: left shift |
| *LXOR | Bitwise logical exclusive OR operator |
| *NOT | Boolean negation operator |
| *NQ | Relational operator: is not equal to |
| O | Octal constant indicator |
| *OFF | Boolean constant stored internally as 0 |
| *ON | Boolean constant stored internally as 1; start delimiter for chronic statement |
| OPTIMIZE | Optimization directive start delimiter |
| *OR | Boolean OR operator |
| *OVERLAY | Storage allocation declarator |
| *PACK | Packing procedure |
| PARAMETER | Data item attribute |
| *PROC | Procedure and function declarator |
| *REM | Remainder function |
| *REMQUO | Remainder-and-quotient procedure |
| *RND | Rounding function |
| *RSH | Logical operator: right shift |
| *,S | Scaling operator |
| *SIC | Start delimiter for optimization exemption directive |
| *SIGN | Sign-determination function |
| SPACE | Storage space optimization directive |
| *START | Start statement; first statement in a CLASP program |
| *STOP | Stop statement; halts program execution |
| T | Textual type declarator |
| TEMP | Data item attribute |
| *TERM | Termination statement; last statement in a CLASP program |
| TEXT | Textual type declarator |
| *THEN | Start delimiter for THEN statement group in con- ditional statement |
| TIME | Execution time optimization directive |
| *TO | Loop statement limit indicator; inhibit/enable statement delimiter |

TABLE III
CLASP PRIMITIVES
(continued)

An asterisk indicates reserved words that may not be used as programmer-defined identifiers.

| | |
|----------|---|
| *TRACE | Start delimiter for debugging directive |
| *TRUE | Boolean constant stored internally as 1 |
| *UNCOUNT | End delimiter for timing directive |
| *UNLOCK | Enable directive |
| *UNPACK | Unpacking function |
| *UNPACX | Sign-extended unpacking function |
| *UNSIIC | End delimiter for optimization exemption directive |
| *UNSPACE | End delimiter for storage space optimization directive |
| *UNTIME | End delimiter for execution time optimization directive |
| *UNTRACE | End delimiter for debugging directive |
| X | Hexadecimal constant indicator |

by double apostrophes at its beginning and end, and is ignored by the compiler. For example

```
IF "CURRENT" RANGE GR "GREATER THAN MAXIMUM"  
    LIMIT
```

shows two comments embedded in one statement. Since comments are interpreted as blanks, to the compiler the statement reads

```
IF RANGE GR LIMIT
```

Numbering conventions.--The bits of a word are numbered consecutively from left to right, the first such bit (the sign bit) being bit 0 and the last being the least significant. Similarly, six bytes of text would be numbered 0 to 5, with the leftmost being byte 0 and the rightmost, byte 5.

Each dimension of an array is numbered consecutively from 0 through the number of elements in the dimension minus 1.

All numbers are interpreted to be decimal numbers unless otherwise indicated by the programmer.

Sample program.--A sample CLASP source program listing is reproduced on page 20.

SAMPLE PROGRAM

START

```

'' THIS IS A CLASP PROGRAM TO PERFORM NUMERICAL INTEGRATION USING THE
TRAPEZOIDICAL RULE. THE PROGRAM CALLS FUNCTION TRAPINT WHICH RETURNS
THE VALUE OF THE INTEGRAL AND STORES IT IN ITEM RESULT. ''

```

```

    DECLARE FIXED,ORDINATE(10) 24 10,
                      RESULT    24 8

```

```

    DECLARE BOOLEAN,ERROR =FALSE
    DECLARE TEXT,MESS1 = '

```

DIRECT

```

'' A DIRECT CODE PROCEDURE TO INPUT 10 SAMPLE POINTS FROM A
FUNCTION INTO ARRAY ORDINATE SHOULD APPEAR HERE,ALTHOUGH
IT WILL NOT BE CODED ''

```

END

```

    RESULT = .TRAPINT(ORDINATE,10,0.0125A12)

```

```

    IF ERROR

```

```

    THEN MESS1 = 'ERROR IN INTEGRATION'

```

```

    END

```

```

    STOP

```

```

    PROC .TRAPINT(Y,N,H)

```

```

    DECLARE INTEGER,I,N

```

```

    DECLARE FIXED, Y(N) 24 10, ''THE ARRAY OF ORDINATES''

```

```

                      H 24 12, ''THE STEP SIZE OF THE ABSCISSA''

```

```

                      TRAPINT 24 8, ''THE VALUE OF THE INTEGRAL''

```

```

                      PARTIAL TEMP ''A LOCAL VARIABLE TO THE PROC''

```

```

    IF N LS 2 THEN ERROR = TRUE $ GOTO EXITINT END

```

```

    PARTIAL = (0.5 * (Y(0) + Y(N-1))) .S(24,10)

```

```

    N = N - 2

```

```

    FOR I = 1 TO N

```

```

        PARTIAL = PARTIAL + Y(I)

```

```

    END

```

```

    TRAPINT = PARTIAL * H

```

```

EXITINT. EXIT

```

```

TERM

```

DATA DECLARATION

The data environment in which a CLASP program operates is described by means of data declarations. Data may be represented in two forms, as constants and as data items. A constant is implicitly declared as to its type and other attributes by its occurrence in the executable part of the program. A data item must be explicitly declared at the beginning of the program or procedure in which it is used; the data declaration gives it an identifier which allows referencing it, and also assigns to it a type to indicate to the compiler how it should be manipulated in subsequent processing. Constants and data items can contain fixed-point, integer, Boolean, or textual information. Floating point is also allowed for computers having hardware floating-point arithmetic commands.

Aggregations of several similar and related data items may be declared as arrays in which individual items or groups of items are referenced by the use of the array identifier and appropriate subscripts or implicitly declared subscripts. Group declarations may be used to define groups of data items as having a common identifier as well as independent identifiers for each separate item. Index declarations are provided to permit the efficient allocation of machine registers to subscripts. Finally, declarations controlling hardware assignments and memory allocation are provided.

Fixed-Point Constants

Fixed-point constants are used to specify the numeric values of unchanging and known decimal numbers that are used by the CLASP program. Fixed-point constants must be expressed in decimal and may have an optional sign, an exponent, and a fractional bit specification.

Format

$$\begin{array}{l} d_1. \\ \{ \pm \} . d_2 \quad \{ E \{ \pm \} d_3 \} \{ A \{ \pm \} d_4 \} \\ d_1 . d_2 \end{array}$$

The d's represent strings of decimal digits. d_1 and d_2 are, respectively, the integer and fractional parts of the constant. The optional E indicates that the constant has an exponent. The $\{ \pm \} d_3$ following the E is the power of 10 by which the constant is multiplied (for example, 3000, which can be expressed in scientific notation as 3.0×10^3 , can be expressed in CLASP as 3.0E3). The optional A indicates that the constant has a fractional bit specification. A minimum of $\{ \pm \} d_4$ bits will be allocated by the compiler to the fractional part of the constant. Note that no blanks may appear within a constant.

If the number of fractional bits is not specified, the compiler will assign a scaling consistent with the context of the constant's use. In general, one computer word will be allocated to a fixed-point constant. If one is used in a context of higher precision operations, its precision will be compatible.

If the number of fractional bits is specified, the compiler will locate the radix point according to the following criteria:

- (1) If d_4 is positive or zero, the radix point will be at least d_4 bits to the left of the low-order end of the word.
- (2) If d_4 is negative, the radix point will be no more than d_4 bits to the right of the low-order end of the word. (Negative fractional bit specifications are explained in more detail under fixed-point declarations, page 34.)
- (3) If d_4 is greater than the computer word size, the radix point will be outside the word to the left.

ExamplesAcceptable

+123.

.0

-.5E-10

3.1416A14

6.02E23A-60

Not Acceptable

52E-6 (no decimal point)

56,974.0 (comma not allowed in constant)

32.E-.5 (decimal point not allowed in exponent)

Floating-Point Constants

Floating-point constant capability is provided for aerospace computers having hardware floating-point arithmetic operations; otherwise fixed-point constants must be used. The floating-point constant, which looks exactly like the fixed-point constant except that a fractional bit specification may not be written, may be distinguished from a fixed-point constant by the context of its use.

Format

$$\begin{array}{c} d_1 \cdot \\ \{ \pm \} \cdot d_2 \quad \{ E \{ \pm \} d_3 \} \\ d_1 \cdot d_2 \end{array}$$

The d's represent strings of decimal digits. d_1 and d_2 are, respectively, the integer and fractional parts of the constant. The optional E indicates that the constant has an exponent. The $\{ \pm \} d_3$ is the power of 10 by which the constant is multiplied. Note that no blanks may appear within a constant.

The number of words allocated to the constant depends upon the floating-point format of the particular implementation.

Integer Constants

Integer constants are used to specify numeric values in those cases where the values are known and unchanging integers. They are most frequently used as indices, as values for setting indices and counters, and as coefficients in equations. An integer constant consists of an optional sign followed by a string of decimal digits.

Format

{₋⁺} decimal-digits

Examples

Acceptable

154
-2095
+1

Not Acceptable

154.0 (decimal point not allowed)
-99E10 (exponent not allowed)
12A7 (fractional bit specification not allowed)

The range of integer constants, an implementation-dependent restriction, is limited to plus or minus the highest memory address of the object computer. However, the ability of fixed-point constants to assume only integer values may be used to enter values of greater absolute magnitude if desired.

Boolean Constants

Boolean constants are used to specify either of two possible Boolean values in a readable form.

Format

| | | |
|------|----|-------|
| TRUE | or | FALSE |
| ON | | OFF |

TRUE and ON are both stored internally as the integer 1; FALSE and OFF, as the integer 0. Whether TRUE or ON (or FALSE or OFF) is used makes no difference in program execution; the choice should be made so as to promote program readability.

Textual constants are used to specify the literal text to be included in the program. They are specified by a string of characters surrounded by primes.

Format

{decimal-integer} 'character-string'

All machine-readable characters are allowed in the string. The optional integer specifies the number of characters in the string and is required only when a prime is one of the characters imbedded in it.

Examples

- (1) '3.14159'
- (2) 'PUSH DESTRUCT BUTTON, THEN HOLD EARS'
- (3) 12 'ERROR 'ABLE''

This is a 12-character constant with imbedded primes.

Binary, Octal, and Hexadecimal Constants

Binary, octal, and hexadecimal constants are used to specify the exact machine bit configurations of values used by the program. They are considered to have the radix point immediately to the right of the last digit written; the radix point itself should not be written. Blanks should not appear within these three types of constants.

Format (Binary Constants)

B'binary-digits'

Each binary digit, at least one of which must appear between the primes, is represented by a 0 or a 1 and is equivalent to one bit of a computer word.

Examples

(1) B'011011'

This is equivalent to the decimal value 27.

(2) B'1111111000000'

Format (Octal Constants)

O'octal-digits'

Each octal digit, at least one of which must appear between the primes, is represented by the numerals 0 through 7 and is equivalent to three bits of a computer word.

Examples

(1) O'1000'

This is the octal equivalent of the decimal value 512.

(2) O'77777776'

Format (Hexadecimal Constants)

X'hexadecimal-digits'

Each hexadecimal digit, at least one of which must appear between the primes, is represented by the numerals 0 through 9 and the letters A through F and is equivalent to four bits of a computer word.

Examples

(1) X'12C'

This represents the hexadecimal equivalent of the decimal value 300.

(2) X'1B3F52'

Location Constants

Location constants permit direct reference to the memory address of a data item or a labeled statement.

Format

L 'item-name'
L 'statement-label.'

Note that statement labels are followed by a period and item names are not.

The location constant becomes identical to the numeric value which represents the memory address of the specified identifier or, for multiple-word items, to the value representing the address of the first word of the specified identifier. If the identifier is modified by a subscript, the value is the address of the array element referenced. Only integer constants are allowed as subscripts in location constants.

Examples

(1) L'ABLE'

This constant represents the memory address of ABLE.

(2) L'BETA(4)'

This constant represents the address of the fifth element of BETA, which is located at BETA+4 if BETA is a single-word array, at BETA+8 if BETA is a double-word array, etc.

(3) L'START1.'

This constant represents the address of the first instruction of the statement labeled START1.

Data declarations are used to define data items with their associated identifiers and to describe the types and attributes affecting their subsequent manipulation. All data items must be declared before any manipulative reference to them.

A single data declaration may be used to describe one or more items of a single data type; the attributes common to a group of items may be factored in the declaration to reduce the amount of writing required.

Format

DECLARE type {attributes}[, identifier {attributes}]₁[∞]

Each identifier represents a data item which is declared as having the type specified. The attributes common to a group of data items being declared with a single statement are written immediately after the type field. The attributes pertaining to a specific item within the common group are written immediately after the identifier of that item. If any conflict exists between the common attribute declaration and the specific item declaration, the latter prevails. The attributes may appear in any order within a factored declaration as well as in specific item declarations.

The type field may be specified:

FIXED or A for fixed-point data

FLOATING or F for the optional floating-point data

INTEGER or I for integer data

BOOLEAN or B for Boolean data

TEXT or T for textual data

Those attributes that are specific to a particular data type are discussed in the description of that type. Those global attributes that are applicable to all data types are the CONSTANT, PARAMETER, and preset value attributes, and are discussed here.

The attribute CONSTANT is used for items whose values do not change during program execution and are known at compilation time. Such values are supplied via a preset value attribute in the data declaration. Recompile is required if the value of a data item declared with CONSTANT is changed.

Data Declarations

The attribute `PARAMETER` is used for items whose values do not change during program execution but may not be precisely known until the object program is loaded. A change in value for a data item declared with `PARAMETER` will not require recompilation if such a change is consistent with the other attributes of the item.

Both the constant and parameter items may be allocated to read-only storage by the compiler. The attribute `CONSTANT` allows the compiler to generate optimum code for handling constant values such as an integer which is a power of 2 and is used as a divisor. If such an item is declared with `CONSTANT`, the compiler will generate a shift instruction that is faster than the more general divide instruction that would have been used if the item had been treated as a variable or a parameter. If a data item is not declared with `CONSTANT` or `PARAMETER`, it is assumed to be a variable, changeable during program execution and allocated to read/write memory.

A preset value must appear if the attribute `CONSTANT` is used. The effect of the preset value attribute is to preset data items to a constant value determined by a specified formula.

Format (Preset Value Attribute)

= formula

Preset formulas for numeric declarations are operands separated by the operators `+`, `-`, `*`, `/`, or `**`. Evaluation of the formulas is carried out in the same manner as that of ordinary statement formulas. Parentheses may be used to clarify the order of evaluation. The operands must be CLASP numeric constants or item names which have previously been declared `CONSTANT` with a preset value. Boolean and textual items may be preset only with Boolean and textual constants, respectively.

Declaring a numeric item with `CONSTANT` permits the programmer to symbolically reference the item's value in any succeeding preset formulas, providing ease of program modification when changing identical preset values used several times. All preset formulas are evaluated at compile time; no object code is generated for them.

Example (Preset Value)

```
DECLARE INTEGER, M CONSTANT = 24, K CONSTANT = 48,  
      L = (M+K)/12, TWO10 = 2**10
```

M and K are declared as constant integer items preset to 24 and 48, respectively. L is declared as an integer item preset to $(24+48)/12 = 6$. TWO10 is declared as an integer item preset to $2^{10} = 1024$.

Fixed-Point Declarations

The fixed-point data declaration specifies the criteria to be used by the compiler in allocating fixed-point data items to memory and using them in computations. Whenever the data declaration is such as to give the compiler a choice of possible radix point locations, it will choose that location which minimizes the amount of rescaling necessary to perform calculations with the item.

The format of the fixed-point data declaration is similar to that of the general data declaration, with the addition of specific attributes peculiar to this type.

Format

$$\text{DECLARE } \overset{\text{FIXED}}{\underset{\text{A}}{\text{A}}} \{ \{n_1\} n_2 \} \left\{ \begin{array}{c} \text{CONSTANT} \\ \text{PARAMETER} \\ \text{TEMP} \end{array} \right\} \{ =\text{formula} \}$$

$$\left\{ , \text{identifier } \{ \{n_1\} n_2 \} \left\{ \begin{array}{c} \text{CONSTANT} \\ \text{PARAMETER} \\ \text{TEMP} \end{array} \right\} \{ =\text{formula} \} \right\}_1^{\infty}$$

n_1 and n_2 are integers indicating, respectively, the minimum number of total bits to be allocated to the item and the minimum number to be reserved for the fractional part; the compiler locates the radix point for the item according to the values specified. The CONSTANT and PARAMETER attributes are as discussed in the previous section. The TEMP attribute is used to prevent the compiler from rescaling items, that is, when it is desired to employ an item for temporary storage of fixed-point values having many different scalings. When a TEMP item appears as an element of a formula, it is assumed to have the same scaling as the closest previous formula to which the variable specified by the declaration's identifier was assigned. (TEMP is discussed further in the section on fixed-point assignment.)

If n_1 and n_2 are not specified, the compiler will allocate one full word to the item, with no fractional bits. If only one integer appears in the declaration, it will be considered to be n_2 , and n_1 will be assumed to be the computer's word size. If both n_1 and n_2 appear in the declaration, n_1 must be coded first.

The fractional bit specification, n_2 , may be negative or greater than n_1 , indicating a radix point outside the number of bits. The number of words, m , allocated to a fixed-point item will be the greatest integer not exceeding $(n_1 + \pi - 1)/\pi$, where π is the number of bits per computer word. If the bits in the item are ordered from the left as 0 to the right as $m \cdot \pi - 1$, this numbering scheme may be extended by assigning -1, -2, -3, etc., to the imaginary bit positions to the left of the item and $m \cdot \pi$, $m \cdot \pi + 1$, $m \cdot \pi + 2$, etc., to the imaginary bit positions to the right of the item. The radix point may be placed at any position from the right of bit $n_1 - n_2 - 1$ to the right of bit $m \cdot \pi - n_2 - 1$. In the case where $n_1 = m \cdot \pi$, the radix point will be placed at position $n_1 - n_2 - 1$. Thus if it is desired to fix the location of the radix point, the total bit specification, n_1 , must be a multiple of the computer word size, π .

It is important that the number of bits declared via n_1 and n_2 do not exceed the number actually required to handle the expected range and the required precision for any item. Specifying only the necessary number will provide the compiler with the flexibility to choose a set of scalings that meets the declared minimum requirements and at the same time minimizes manipulation of data items. For example, assume that for a specific variable the largest magnitude is known to be 30,000 and the allowable tolerance of data representation is 0.05. It takes at least 15 bits to represent the integer portion (16,383 is the largest integer representable with 14 bits; 32,767 can be represented with 15 bits) and at least 5 bits for the fractional part (.03125 is the smallest non-zero fraction which can be represented with 5 bits). All numeric quantities are signed, so that 1 bit is required for the sign. The declaration might be:

```
DECLARE FIXED, VAR 21 5
```

Thus VAR is declared as a fixed-point item having 21 total bits (15 integer + 5 fractional + 1 sign) and 5 fractional bits. Assuming a 24-bit word, 3 bits will be left over. The compiler may locate the radix point after bits 15, 16, 17, or 18.

If the total number of bits is omitted from the data declaration, a full word is assumed necessary to represent the quantity. In this case, the compiler would have no choice but to place the radix point after bit 18.

Fixed-Point Declarations

Examples (assume 24-bit word size)

- (1) DECLARE A PARAMETER,
 RADIUS -4 CONSTANT = 1.6032788E8,
 ORBIT,
 KACCEL 15 36 = 4.6571E-5

RADIUS is declared as a fixed-point constant (overriding the factored PARAMETER attribute) preset to 1.6032788E8. A full word is required for this item (since the total bit specification is missing), with -4 bits reserved for the fractional part. The radix point will be located 4 bits to the right of the computer word. ORBIT is declared as a full-word fixed-point parameter with no fractional bits. KACCEL is declared as a fixed-point parameter preset to 4.6571E-5 (this may be a nominal value which can be changed at load time). Fifteen total bits are required, with at least 36 reserved for the fractional part. The compiler may place the radix point anywhere from 12 to 21 bits to the left of the word.

- (2) DECLARE FIXED TEMP, RE1, RE2, RE3, RE4

RE1, RE2, RE3, and RE4 are declared as fixed-point items having the TEMP attribute. No scaling information is given in the declaration, since it will be determined from formulas assigned to them in the program. Single words are allocated for each item.

- (3) DECLARE FIXED 48 0 TEMP, RE1, RE2, RE3, RE4

This is similar to Example (2) but includes bit specifications because double-precision TEMP items are required. However, the TEMP attribute tells the compiler to ignore scaling information derived from the bit specification.

Floating-Point Declarations

The floating-point data declaration is used to describe floating-point items when the hardware capability is provided. It is similar to the fixed-point declaration except that a precision specification replaces the total and fractional bit specifications.

Format

$$\text{DECLARE } \begin{matrix} \text{FLOATING} \\ \text{F} \end{matrix} \{ \text{precision} \} \left\{ \begin{matrix} \text{CONSTANT} \\ \text{PARAMETER} \end{matrix} \right\} \{ = \text{formula} \}$$

$$\left\{ , \text{identifier } \{ \text{precision} \} \left\{ \begin{matrix} \text{CONSTANT} \\ \text{PARAMETER} \end{matrix} \right\} \{ = \text{formula} \} \right\}_1^{\infty}$$

The precision specification is a decimal number indicating the number of decimal places required.

The compiler will allocate the item to a single computer word (single precision) or to two computer words (double precision) based on the precision specified. If not specified, single precision is assumed.

Examples

(1) DECLARE F, DISTANCE, SPEED

DISTANCE and SPEED are declared to be single-precision floating-point items.

(2) DECLARE FLOATING, ABLE 7

Item ABLE will have at least seven decimal digits precision in its floating-point representation.

Integer Declarations

Integer data items are used for those quantities which can assume only integer values less than a certain limit, for example, counters, program logic path indicators, and indices for arrays. They correspond numerically to similarly declared fixed-point items with zero fractional bits. To maintain object code efficiency, the integer declaration should be used where possible instead of the more general fixed-point declaration in which the size of the item is controlled by the declaration. The largest allowable magnitude of an integer item is equal to the highest memory address of the computer; this limiting size is implicit in the declaration. Although integer items are allocated to a full word (which may be bigger than the maximum allowable integer size), integer calculations may be truncated from the high-order end of the allowable size owing to the allocation of integer data to integer-sized registers if the computer has them. Thus unpredictable results may occur if integers are allowed to exceed the maximum value.

Format

$$\begin{array}{c} \text{DECLARE} \quad \text{INTEGER} \left\{ \begin{array}{l} \text{CONSTANT} \\ \text{PARAMETER} \end{array} \right\} \{=\text{formula}\} \\ \quad \quad \quad \text{I} \\ \left\{ \begin{array}{l} \text{, identifier} \end{array} \right\} \left\{ \begin{array}{l} \text{CONSTANT} \\ \text{PARAMETER} \end{array} \right\} \{=\text{formula}\} \left. \vphantom{\begin{array}{l} \text{CONSTANT} \\ \text{PARAMETER} \end{array}} \right\}^{\infty}_1 \end{array}$$

Examples

(1) DECLARE I = 0, K, L, M = 1, N

K, L, and N are declared as integer items preset to 0, and M as an integer preset to 1.

(2) DECLARE INTEGER CONSTANT, CL1 = 9, CL2 = -100

CL1 and CL2 are declared as integers with constant values of 9 and -100, respectively.

Boolean data items are used chiefly in program logic as path indicators and in input/output routines as signals of a Boolean status. Boolean items can take on two states: TRUE (or ON) and FALSE (or OFF), represented internally as the integers 1 and 0, respectively. A full word is allocated to these items, which use only the global attributes.

Format

$$\begin{array}{c} \text{DECLARE} \quad \text{BOOLEAN} \quad \left\{ \begin{array}{c} \text{CONSTANT} \\ \text{PARAMETER} \end{array} \right\} \quad \{ = \text{formula} \} \\ \quad \quad \quad \text{B} \\ \left\{ \begin{array}{c} \text{, identifier} \left\{ \begin{array}{c} \text{CONSTANT} \\ \text{PARAMETER} \end{array} \right\} \quad \{ = \text{formula} \} \end{array} \right\}_1^{\infty} \end{array}$$

Example

```
DECLARE BOOLEAN, ORBIT, GO CONSTANT = TRUE,
      NOGO CONSTANT = FALSE, OVFLO = 0
```

ORBIT is declared as a Boolean item; GO and NOGO are declared as constant Boolean items having the values TRUE and FALSE, respectively; OVFLO is declared as a Boolean item preset to 0 (FALSE).

Textual Declarations

Textual data items are used to contain alphabetic, numeric, and special character strings in the character code of the computer for which the program is written. Thus they provide for the inclusion and manipulation of non-numeric quantities, for example, a keyboard input signal that must be decoded or an error message to be sent to a pilot display.

Format

$$\begin{aligned} & \text{DECLARE } \begin{matrix} \text{TEXT} \\ \text{T} \end{matrix} \{ \text{number-of-bytes} \} \left\{ \begin{matrix} \text{CONSTANT} \\ \text{PARAMETER} \end{matrix} \right\} \{ =\text{formula} \} \\ & \left\{ , \text{identifier} \{ \text{number-of-bytes} \} \left\{ \begin{matrix} \text{CONSTANT} \\ \text{PARAMETER} \end{matrix} \right\} \{ =\text{formula} \} \right\}_1^\infty \end{aligned}$$

The number-of-bytes attribute is an integer indicating the number of characters reserved for the item.

If the number of bytes is not declared and a preset value is present, the item will have the same number of bytes as the preset value. If both the number of bytes and the preset value are missing, the number of bytes per computer word will be reserved.

Example

```
DECLARE TEXT, MESS1 80, MMI = 'PUSH START'
```

MESS1 is declared as an 80-byte textual item. MMI is declared as a 10-byte textual item because its preset value, 'PUSH START', is 10 bytes long.

Array declarations are used to define groups of items having a related nature and common attributes. The items are arranged such that the group may be referenced by an identifier which designates the entire array and the individual elements within the array may be referenced by subscripts. Each element of an array has the type and attributes given in the declaration. The format for the declaration is an extension of that for single item declarations.

Format

DECLARE type {attributes}

{(integer-constant{, integer-constant}²₀)}{=constant-list}

{, identifier {(integer-constant{, integer-constant}²₀)}}

{attributes}{=constant-list}[∞]₁

The parenthesized list of integer constants (the dimension list) indicates that the item being declared is an array. The identifier is the array name, and the constant list is the mechanism for presetting the elements of the array. The dimension and constant lists are treated like any other attributes and thus may be written factored or may be written uniquely for each identifier.

Arrays can have one, two, or three dimensions. The value of the first integer in the dimension list is the number of rows; that of the second, the number of columns; and that of the third, the number of planes. The number of elements in the array is the product of all dimensions in the list, and the number of words reserved for the array is the number of words per element times the total number of elements.

If the dimension list appears in the factored part of the array declaration, it applies to all arrays declared. If it appears to the right of an identifier, it applies to that identifier only, and will override a factored declaration.

Individual elements in an array may be preset as desired using the constant list; its format is an extension of that for presetting simple items.

Array Declarations

Format (Constant List)

$$= \{ (\{ \text{repetition-factor } () \text{ formula } () \} \} \\ \left\{ , \{ \text{repetition-factor } () \text{ formula } () \} \right\}_0^{\infty} () \}$$

The optional repetition factor is an integer indicating the number of times the following formula's value is to be repeated in the constant list. The pair of parentheses surrounding the formula is required only when a repetition factor is specified, and the pair of parentheses surrounding the entire list is required only when a comma appears inside the list. The formulas are of the same format as those used in the preset value attribute.

Beginning at the first, each element in the array is preset to the corresponding value in the constant list. For one-dimensional arrays, the correspondence between the constant list and the elements is obvious. For two- and three-dimensional arrays, the preset values are stored in the order of the array's layout in memory. All arrays are stored linearly in memory, so that the first subscript of an array varies most rapidly, the last subscript, least rapidly. Thus, two-dimensional arrays are stored by columns and three-dimensional arrays by planes with the planes stored by columns.

Examples

- (1) The following array, named MAT and having the indicated preset values

| | | Columns | | | |
|------|---|---------|---|---|----|
| | | 0 | 1 | 2 | 3 |
| Rows | 0 | 0 | 3 | 7 | 7 |
| | 1 | -6 | 4 | 7 | 59 |
| | 2 | -48 | 6 | 7 | 11 |

is declared by:

```
DECLARE INTEGER, MAT (3, 4)
      = (0, -6, -48, 3, 4, 6, 4(7), 59, 11)
```

Each element of the 3 × 4 array holds an integer value. The value 7 is given a repetition factor of 4. The constant list presets the elements of MAT in columns, The array's memory arrangement will be:

```
MAT(0, 0) = 0
MAT(1, 0) = -6
MAT(2, 0) = -48
MAT(0, 1) = 3
MAT(1, 1) = 4
MAT(2, 1) = 6
MAT(0, 2) = 7
MAT(1, 2) = 7
MAT(2, 2) = 7
MAT(0, 3) = 7
MAT(1, 3) = 59
MAT(2, 3) = 11
```

```
(2) DECLARE FIXED (3), VELOC 17 2, ACCEL 12 5,
      POS 48 6 = 3(0.0), XFORM(9) 24 10
```

VELOC, ACCEL, and POS are each declared as three-element fixed-point arrays, POS being preset to 0. XFORM is declared as a nine-element fixed-point array, overriding the factored three-element array specification.

```
(3) DECLARE T, ABC (4, 5, 3) 20
```

The array ABC has three dimensions of four rows, five columns, and three planes. Each element contains 20 characters of textual information.

Variable-Dimension Array Declarations

Variable-dimension arrays are used to allow the writing of a procedure that has several arrays as actual parameters, the individual array having different dimensions. For example, a matrix inversion procedure may be developed to operate on an $N \times N$ fixed-point matrix, where N is a variable that is supplied to the procedure. N is fixed for any particular matrix that is to be inverted but is different for different matrices. A variable-dimension array declaration does not result in the allocation of memory locations; it provides the mechanism by which an array identifier without subscript value limitations can be used in writing a procedure and by which linkages can be made to fixed-dimension arrays that are allocated.

A variable-dimension array is specified by replacing one or more of the integer constants in the dimension list of an array declaration with the names of simple integer items. The size of the variable dimension is the value of the integer at procedure execution time. The number of dimensions may not vary.

Format

$$\text{DECLARE type \{attributes\} } \left\{ \begin{array}{l} \text{identifier } \{, \text{identifier } \}^2 \\ \text{constant } \{, \text{constant } \}^0 \end{array} \right\}$$

$$\left\{ , \text{identifier } \begin{array}{l} \text{identifier } \{, \text{identifier } \}^2 \\ \text{constant } \{, \text{constant } \}^0 \end{array} \text{ \{attributes\} } \right\}_1^{\infty}$$

The identifiers and constants in the dimension list must all be of integer type. If the identifiers in the dimension list have not been previously declared, they will automatically be declared as integer.

Example

```
DECLARE FIXED, TABLE (I, J, K)
```

TABLE is declared as a fixed-point array having three dimensions of size I, J, K. Assuming this declaration is made in a procedure heading, I, J, K would be passed as input parameters of the procedure; TABLE would be either an input or output parameter.

Implicit subscripts are used to minimize the writing required when referencing an array by including in the array's dimension list the names of integer variables that are to be implicitly defined as subscripts for that array.

Format

```

DECLARE type {attributes}

      {(identifier constant {, identifier constant}20) }

      { , identifier {(identifier constant

      {, identifier constant}20) } {attributes} }1∞
  
```

The identifiers and constants in the dimension list must be of integer type. If the identifiers in the dimension list have not been previously declared, they will automatically be declared as integers.

Example

```
DECLARE FIXED, ABLE (R 3, C 3)
```

ABLE is declared as a 3 x 3 fixed-point array whose row and column subscripts will be R and C, respectively, whenever the array name is used without subscripts.

The use of implicit subscripts is discussed in more detail in the section on formulas and assignment statements.

Group Declarations

Group declarations are used to name groups of single data items such that a group as a whole can be referenced by its group name while its individual elements may be referenced by their particular names. The elements of a group can have different attributes; the commonality necessary for arrays is not required.

A group is declared by labeling a data declaration statement.

Format

identifier. data-declaration

The group name is indicated by the identifier, which must be followed by a period. The data declaration statement can be of fixed, floating (if available), integer, Boolean, or textual type.

Examples

(1) ABLE. DECLARE INTEGER, A, B, C, D

Group ABLE consists of four integers.

(2) BAKER. DECLARE FIXED 24 10, E, F, G(10)

Group BAKER has three fixed-point items, one of which is a 10-element array.

The index declaration is used to indicate to the compiler that a particular integer data item should be maintained, when possible, in the hardware registers most suitable for holding and utilizing array subscripts instead of in the memory locations normally used for the local area of the program in which the item is used. Effective use of the index declaration thus increases object code efficiency via optimum utilization of the limited number of suitable registers. Only items that are frequently employed as array subscripts should be defined with the index declaration; otherwise, little or no benefit will accrue from its use.

Format

DECLARE INDEX {, identifier}₁[∞]

The identifiers, if previously declared, must be of integer type. If they have not been previously declared they will automatically be declared as integer items.

Example

DECLARE INDEX, I, J, K

I, J, K are declared as integers and will be maintained in index registers with a higher priority than other subscripts not so declared.

Hardware Declarations

The hardware declaration is used to specify the interrelationship between the program and the object computer's hardware configuration. By allowing direct reference to machine registers normally invisible to the applications programmer, it can be used to promote object code efficiency. Unlike the index declaration, which merely establishes priorities for usage of hardware registers, the hardware declaration results in definite allocations in the resulting object program.

Hardware declarations are machine dependent; the CLASP capability is an outline of their use in a general form which avoids specifying details peculiar to any particular computer.

Format

```
DECLARE HARDWARE type {attributes}
```

```
{, identifier {attributes}=hardware-code}1∞
```

The identifier is the name assigned to a machine register, and the hardware code is the machine address or name of the register. The item description for the identifiers, which may be factored, indicates the description for the data which are associated with the hardware registers.

Different identifiers, types, or attributes may be associated with the same hardware code to allow the same register to assume different data types or attributes and to be referenced by different names.

Hardware declarations should be used in conjunction with the inhibit/enable statements LOCK and UNLOCK to avoid the possibility that the compiler may overwrite a value that the programmer wishes to save.

Example

```
DECLARE HARDWARE I, INDEX1 = 2, INDEX2 = 3  
DECLARE HARDWARE A, VAL 16 = 2
```

Hardware Declarations

The hardware declarations demonstrate assigning names to some of the 16 registers of the IBM 360. INDEX1 and INDEX2 are declared to be integers, whereas VAL is declared fixed point with a 16-bit fractional part. Note that INDEX1 and VAL refer to the same general-purpose register.

Overlay Declarations

The overlay declaration is used to indicate the order in which previously declared simple items or arrays should appear in the object program and the location in which they should be stored in memory. It is also used to assign two or more such items or arrays to the same memory location, enabling economical use of storage to be made or a single data word to be treated as having different data types or attributes.

Format

$$\text{OVERLAY } \begin{matrix} \text{constant} \\ \text{identifier } \{, \text{identifier} \} \end{matrix} \begin{matrix} \infty \\ 0 \end{matrix} \left\{ \begin{matrix} =\text{identifier} \{, \text{identifier} \} \\ 0 \end{matrix} \right\} \begin{matrix} \infty \\ 0 \end{matrix}$$

The constant (which may be integer, hexadecimal, octal, or binary) indicates an absolute location in core. The identifiers are the names of previously declared simple items, arrays, or subscripted items with constant value subscripts.

When a constant is used following the primitive OVERLAY, the declaration is an absolute overlay: the items represented by the identifiers to the right of the equals sign will start at the specified absolute location in core and will be in the order given in the declaration.

When a series of identifiers is used immediately after OVERLAY and there is no equals sign or second series of identifiers, the items will be allocated to memory in the order given.

When an identifier or series of identifiers is used immediately after OVERLAY and is followed by an equals sign and another series of identifiers, the items to the right of the equals sign will overlay those to the left in the order given. That is, the first item on the right will overlay the first on the left; the second item on the right will overlay the second on the left; etc. If the series on the right and left do not consist of the same number of items, the excess items will be allocated immediately after the last one for which a match exists, and in the order given.

If a subscripted item is used in an overlay declaration, the entire array is positioned so that the specified element of the array satisfies the overlay declaration. Any item preceding or following a subscripted item in an overlay declaration will be allocated in storage so that it immediately precedes or follows, respectively, the specified element of the array. If an array name is

used without a subscript, the compiler will position the array's first element in the indicated position and any identifiers following the array name will be positioned after the last element of the array.

Identifiers may be repeated within a single overlay declaration or in successive overlay declarations so long as the repeated use of the identifier does not lead to an ambiguous memory address determination for any item. If two or more items have been overlaid in the same storage location, pre-setting them with different values is not permitted. Also, if an item declared with the attributes `CONSTANT` or `PARAMETER` is overlaid by another item, neither of the items may receive a value from an assignment statement or a procedure call.

An overlay declaration may not reference items which have not previously been declared. Thus, the best location for overlay declarations is between the data declarations and the imperative statements of the main program. When data local to a procedure are to be overlaid, the overlay declaration must of course be within the procedure.

Examples

- (1) `DECLARE FIXED, AA(20), BB(10,2), CC(5,2,2)`
`OVERLAY AA = BB = CC`

The overlay declaration causes storage to be allocated so that the three arrays all occupy the same 20 storage locations.

- (2) `DECLARE I, AA, BB, CC, DD, EE, GG`
`OVERLAY AA, BB = CC, DD, EE = CC, DD, GG`

AA and CC occupy the same storage location, followed by the location which BB and DD share. Following this location is the cell which EE and GG jointly occupy. However, the overlay declaration

`OVERLAY AA, BB = CC, DD, EE = DD, GG`

would be illegal because DD is ambiguously assigned to the location AA occupies and the one following it.

Overlay Declarations

- (3) DECLARE BOOLEAN, TAB1, TAB2, TAB3, TAB4, TAB5
OVERLAY TAB1, TAB2
OVERLAY TAB3, TAB4
OVERLAY TAB2, TAB5

In storage, TAB2 follows TAB1 and TAB5 follows TAB2. TAB4 follows TAB3, which are both located in storage independently of TAB1, TAB2, and TAB5.

- (4) DECLARE FIXED, R(10), RR(5), AA, BB, CC, DD, EE, FF
OVERLAY AA, R(5), EE = BB, CC, DD, RR, FF

The overlay declaration generates the following storage structure (identifiers which occupy the same location are on the same line):

| | | |
|------|----|-------|
| R(0) | | |
| R(1) | | |
| R(2) | | |
| R(3) | | |
| R(4) | AA | BB |
| R(5) | | CC |
| R(6) | EE | DD |
| R(7) | | RR(0) |
| R(8) | | RR(1) |
| R(9) | | RR(2) |
| | | RR(3) |
| | | RR(4) |
| | | FF |

Because R(5) follows AA and EE follows R(5), R(4) overlays AA and EE overlays R(6); the entire array is positioned by positioning one element of it. FF follows the last element of RR because RR was specified without a subscript. If RR(0) replaced RR in the overlay declaration, FF would follow RR(0) instead of RR(4).

- (5) DECLARE FIXED, SW(6) 24 0, DW(3) 48 0
OVERLAY SW = DW

On a computer with a 24-bit word size, array SW would be composed of six single-word elements and DW of three double-word elements. Both would occupy the same storage locations, every two elements of SW overlaying one element of DW.

FORMULAS AND ASSIGNMENT

A fundamental program operation is the computation of a new value for a variable and its substitution for a previous value. The computations to be performed are specified by formulas indicating the relationships between items, and the substitutions are done with assignment statements.

The items appearing in formulas may be constants or variables created and described as outlined in the preceding section on data declaration. The identifier used in an item's declaration provides a means for referencing a simple item in a formula. Subscripted, nonscalar, and implicitly subscripted items provide a means for referencing individual elements or groups of elements in arrays. These more complex types of items are further described here.

Formulas are categorized by the type of operations to be performed; these operations are specified by using appropriate operators. Numeric operators indicate arithmetic functions such as addition and multiplication; logical operators indicate bit-by-bit logical operations such as AND and OR; and Boolean operators indicate the comparison between operands such as GREATER THAN and EQUAL TO. Both numeric and logical formulas yield some numeric value or a particular bit configuration. Boolean formulas yield one of two values, TRUE or FALSE. There are no textual operators in CLASP; thus a textual formula can only be a textual variable or constant.

Assignment statements can be of simple form in which a single new value replaces a single old value, or multiple and nonscalar assignment statements may be used, one such statement indicating that several assignments are to be made. In addition, an exchange assignment statement may be used to interchange the values of either scalar or nonscalar variables. A scaling operator is provided such that intermediate and final scalings normally determined automatically during formula evaluation or before assignment can be altered.

Subscripted Items

Subscripts are used to reference elements of a previously declared array.

Format

identifier (integer-formula {, integer-formula }₀²)

The identifier must be a previously declared array, and the number of integer formulas must equal the number of dimensions with which the array was declared.

The integer formula, called the subscript, must be of the following form:

$$v \left\{ \begin{array}{l} c_1 \\ / \\ c_2 \end{array} \right\} \left\{ \begin{array}{l} + \\ - \end{array} \right\} c_3$$

where v is a simple integer variable and the c 's are integer constants.

The value of the subscript is the constant c_1 or the value of v optionally multiplied or divided by c_2 and/or optionally incremented or decremented by c_3 .

The subscript indicates the position of the element of the array being referenced along the dimension to which it corresponds. Since data structures normally begin with a 0, the first meaningful value for a subscript is 0. The last meaningful value is the number of elements in the corresponding dimension minus 1. Thus, a five-element array would have subscripts ranging from 0 to 4.

A diagnostic will be issued by the compiler if a subscript is referenced out of its declared bounds by a constant. However, no detection of an out-of-bounds reference will be made during the execution of the object program. Where data have been located adjacent to an array, it is possible to reference the array out-of-bounds to access the adjacent data, although this is not good coding practice.

ExamplesThis subscripted item is:

- | | |
|-------------------|--|
| (1) INFO(0, 0, 0) | the first element of array INFO |
| (2) INFO(I, 0, 0) | the Ith element of array INFO |
| (3) SPEED(K*3-1) | the element of array SPEED whose subscript is the value of K times 3 minus 1 |
| (4) DISCR(M/3, 2) | the element of array DISCR located at row M/3, column 2. Truncated division is performed; thus if M is 5, row 1 is being referenced. |

Nonscalar Items

Nonscalar subscripts are used to reference an entire row, column, or plane of a previously declared array, or any combination of row, column, and plane, including the entire array. The nonscalar subscript indicates that all possible values of the subscript, as determined by the array declaration, are intended. Nonscalar items may be used in assignment and expression operations as described elsewhere in this section to extend their scope over the entire range of any or all subscripts.

Format

identifier (*integer-formula {, integer-formula }²)₀

As with ordinary subscripted items, the identifier must be a previously declared array, the number of subscripts (integer formulas) or nonscalar subscripts (asterisks) must equal the number of dimensions with which the array was declared, and the integer formulas must be no more complicated than

$$v * c_1 \pm c_2 \text{ or } v / c_1 \pm c_2$$

The nonscalar subscript may be used in one-, two-, and three-dimensional arrays in any or all subscript positions.

Examples

- (1) Assuming that ABLE has been previously declared as the 3×4 array:

$$\begin{bmatrix} -7 & 2 & 10 & 4 \\ 6 & 3 & -1 & 8 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

ABLE(*,*) refers to the entire array; ABLE(0,*) to the row vector:

$$[-7 \quad 2 \quad 10 \quad 4]$$

and `ABLE(*, 2)` to the column vector:

$$\begin{bmatrix} 10 \\ -1 \\ 2 \end{bmatrix}$$

- (2) Assuming that `BAKER` has been previously declared as a one-dimensional array, `BAKER(*)` represents all of its elements.

Implicitly Subscripted Items

If implicit subscripts have been included in an array declaration, any subsequent use of the array identifier without subscripts implies the previously declared subscripts such that the same element is referenced as that which would be referenced if the implicitly declared subscripts were explicitly written. When the array identifier is used with subscripts, all implicitly declared subscripts are overridden. In using the override feature, the number of subscripts written must be the same as the number of dimensions in the array, i.e., any implicit subscripts that are not being overridden must be written. An implicitly subscripted item always is a single element of an array and may not be a nonscalar quantity.

Examples

Assuming that the following implicit subscript declaration has been made:

```
DECLARE, ABLE (R 3, C 3)
```

(1) ABLE

ABLE(R, C) is being implicitly referenced.

(2) ABLE(*, K)

The implicit subscripts have been overridden and vector column K of ABLE is being referenced.

(3) ABLE(R, L)

Again the implicit subscripts have been overridden; element (R, L) is being referenced. Note that even though R had been declared implicitly, both had to be explicitly written.

Numeric formulas are used to specify the calculations to be performed in solving a problem. A numeric formula is a combination of operands (variables, constants, functions, and other numeric formulas) linked by numeric operators. It must be written on a single line which, of course, can be continued for several cards.

The operands may be data items of any of the following types:

- (1) Floating point (if included in the particular implementation)
- (2) Fixed point
- (3) Integer
- (4) Boolean

This list is arranged according to the relative dominance of the various types. Operands of different types may be mixed in a single formula and, if so, will be converted to the most dominant type present before the formula is evaluated.

The numeric operators and their functions are as follows:

NUMERIC OPERATORS

| <u>Operator</u> | <u>Function</u> | <u>Usage</u> |
|-----------------|-----------------|--|
| + | Addition | $x+y$ indicates the addition of x and y . |
| - (binary) | Subtraction | $x-y$ indicates the subtraction of y from x . |
| - (unary) | Negation | $-x$ indicates the negation of x . |
| * | Multiplication | $x*y$ indicates the multiplication of x and y . |
| / | Division | x/y indicates the division of x by y . Division by 0 is undefined; fractional parts are truncated in integer division. |
| ** | Exponentiation | $x**n$ indicates x raised to the n th power. n must be an integer constant. |

Numeric Formulas

The order of evaluation of a numeric formula is determined by the priority of operations to be performed, by the left-to-right order of operations to be performed in the case of operations having equal priority, and by the use of parentheses.

The priority of operations is:

- (1) Negation
- (2) Exponentiation
- (3) Multiplication and division
- (4) Addition and subtraction

Negation is differentiated from subtraction in that a "-" operator specifies subtraction only when there is something in the formula from which the term following the operator can be subtracted.

Balanced pairs of left and right parentheses are used to indicate that the formula they enclose should be evaluated and the result used as an operand in evaluating the remainder of the formula. Parentheses may be nested; the formula enclosed in the innermost pair will be evaluated first, and so on to the outermost pair. If desired, pairs of parentheses may be used to clarify the order of evaluation even if the priority of operations makes them unnecessary.

Examples

| (1) <u>CLASP Formula</u> | <u>Mathematical Equivalent</u> |
|--------------------------|--------------------------------|
| $(A/B)+C$ or $A/B+C$ | $\frac{A}{B} + C$ |
| $A/(B+C)$ | $\frac{A}{B+C}$ |
| $A/(B/C)$ | $\frac{A}{\frac{B}{C}}$ |
| $(A/B)/C$ or $A/B/C$ | $\frac{\frac{A}{B}}{C}$ |

$$(2) \quad (\text{ALPHA}+2) / (\text{ZLIK} + (\text{PI}**3)) - 3.14 + \text{BETA}$$

The order of evaluation would be:

- (a) $\text{PI}**3$
- (b) $\text{ZLIK} + \text{result of (a)}$
- (c) $\text{ALPHA}+2$
- (d) $(\text{c}) / (\text{b})$
- (e) $(\text{d}) - 3.14$
- (f) $(\text{e}) + \text{BETA}$

By algebraic manipulation, the compiler may rearrange formulas to yield a more efficient scheme of operations. Constant terms in expressions may be combined. For example, the formula

$$3.* \text{ALPHA}/1.\text{E}6$$

which requires two operations and storage for two constants, can be optimized and compiled in the form

$$3.\text{E}-6 * \text{ALPHA}$$

which contains only one operation and requires storage for only one constant.

The programmer can aid in the production of efficient object code by factoring out common terms to reduce the number of operations to be performed. To illustrate, the algebraic formula

$$Ax^2 + Bx + C$$

could be written in CLASP as

$$A * X**2 + B * X + C$$

which requires three multiply operations and two addition operations, or as

$$X * (A * X + B) + C$$

which requires one less multiply operation.

Logical Formulas

Logical formulas are used to specify the bit-by-bit manipulation of data items and consist of operands linked by logical operators. A logical formula yields a 0 or 1 for every bit in the formula's result. The operands themselves remain unchanged by the operators. Thus, for example, "shift of z" means that the value of z is shifted; z itself is not altered. A Boolean formula differs from a logical formula in that it yields a value which can only be 0 (FALSE) or 1 (TRUE) for the entire formula.

The following table identifies the logical operators and their functions. In the usage column of the table, r_i is the i th bit of the result of the formula and the symbols shown below represent the indicated types of operands:

| | |
|--------|---|
| x, y | numeric, textual, and logical formulas (x_i and y_i are the i th bits of x and y , respectively) |
| k | integer numeric formula |
| z | numeric or logical formula |

LOGICAL OPERATORS

| <u>Operator</u> | <u>Function</u> | <u>Usage</u> |
|-----------------|-----------------|---|
| LAND | Logical product | x LAND y indicates the bit-by-bit ANDing of x and y such that for all bits i , r_i is 1 if both x_i and y_i are 1 and is 0 otherwise. |
| LOR | Logical sum | x LOR y indicates the bit-by-bit ORing of x and y such that for all bits i , r_i is 0 if both x_i and y_i are 0 and is 1 otherwise. |
| LXOR | Exclusive OR | x LXOR y indicates the bit-by-bit modulo 2 sum of x and y such that for all bits i , r_i is 1 if x_i and y_i differ and is 0 if they are alike. |

LOGICAL OPERATORS (continued)

| <u>Operator</u> | <u>Function</u> | <u>Usage</u> |
|-----------------|-----------------|--|
| LSH | Left shift | z LSH k indicates an arithmetic left shift of z by k bits. Zeros are propagated into vacated positions on the right. Bits shifted out of the left are lost. This shift corresponds to a multiplication of z by 2^k . |
| RSH | Right shift | z RSH k indicates an arithmetic right shift of z by k bits. The sign bit is propagated into vacated positions on the left and bits shifted out of the right are lost. This shift corresponds to a division of z by 2^k . |

Complex logical formulas with more than one logical operation are evaluated from left to right in the same manner as numeric formulas. Parentheses are allowed to clarify the order of operations; see Example (5). In the absence of parentheses, the logical operators have precedences from high to low as follows:

- (1) LSH and RSH
- (2) LAND
- (3) LOR and LXOR

By considering one operand to be a mask, LAND can be used to turn off bits in the other operand, LOR can be used to turn on bits in the other operand, and LXOR can be used to reverse bits in the other operand; see Examples (1) through (3).

If a logical formula is used in an arithmetic operation, the formula's value is assumed to have its radix point immediately to the right of the least significant bit. Multiplications and divisions by powers of 2, for example, can be accomplished using the LSH and RSH operators; see Example (6).

Logical Formulas

Examples

Assuming for simplicity that items are 6 bits in length:

(1) A LAND B'111100'

results in A with the two low-order bits turned off.

(2) A LOR B'000011'

results in A with the two low-order bits turned on.

(3) A LXOR B'000011'

results in A with the two-low order bits reversed.

(4) B'110100' RSH 2

results in B'111101'.

(5) ABLE LOR BAKER LSH 1 LAND CHARLY

is evaluated as if parentheses had been placed as follows:

ABLE LOR ((BAKER LSH 1) LAND CHARLY)

and would take three steps:

- (a) BAKER LSH 1
- (b) (a) LAND CHARLY
- (c) ABLE LOR (b)

In the above formula, assuming that

ABLE = B'011010'
BAKER = B'011101'
CHARLY = B'111000'

then

(a) $\text{BAKER LSH } 1 = \text{B}'011101'$ $\text{LSH } 1 = \text{B}'111010'$

(b) (a) $\text{LAND CHARLY} = \text{B}'111010'$ $\text{LAND B}'111000'$
 $= \text{B}'111000'$

(c) $\text{ABLE LOR (b)} = \text{B}'011010'$ $\text{LOR B}'111000'$
 $= \text{B}'111010'$

(6) $(Z + Q) \text{ LSH } 2$

results in the effective multiplication of $Z + Q$ by 4. This is an example of imbedding shift operations in numeric formulas.

Boolean Formulas

Boolean formulas are used to construct questions about the results of the evaluation of other formulas; evaluating them corresponds to answering the question, the only possible result being either TRUE or FALSE. The numeric representation of the result will be 1 if TRUE and 0 if FALSE, consistent with the Boolean constants TRUE and FALSE. There are two types of Boolean formulas: simple and complex.

Simple Boolean formulas consist of arithmetic, textual, or logical formulas and the relational operators indicated in the following table. In the usage column of the table, the symbols shown below can represent the indicated types of operands:

| | |
|------|--|
| v, w | numeric, textual, and logical formulas |
| x, y | numeric formulas |

RELATIONAL OPERATORS

| <u>Operator</u> | <u>Algebraic Equivalent</u> | <u>Usage</u> |
|-----------------|-----------------------------|---|
| EQ | = | v EQ w is TRUE if v equals w and is FALSE otherwise. |
| NQ | ≠ | v NQ w is TRUE if v does not equal w and is FALSE otherwise. |
| GR | > | x GR y is TRUE if x is greater than y and is FALSE otherwise. |
| GQ | ≥ | x GQ y is TRUE if x is greater than or equal to y and is FALSE otherwise. |
| LS | < | x LS y is TRUE if x is less than y and is FALSE otherwise. |
| LQ | ≤ | x LQ y is TRUE if x is less than or equal to y and is FALSE otherwise. |

Examples (Simple Boolean Formulas)

- (1) FLAG GR 0
- (2) RANGE LQ DIST * SAFETY
- (3) BITS LSL 3 NQ B'101111'

Complex Boolean formulas consist of simple Boolean formulas or items linked by Boolean operators. The operators and their usage are indicated in the following table.

BOOLEAN OPERATORS

| <u>Operator</u> | <u>Usage</u> |
|-----------------|--|
| AND | x AND y is TRUE if both x and y are TRUE and is FALSE otherwise. |
| OR | x OR y is FALSE if both x and y are FALSE and is TRUE otherwise. |
| NOT | NOT x is TRUE if x is FALSE and is FALSE if x is TRUE. |
| EQUIV | x EQUIV y is TRUE if x and y are both TRUE or both FALSE and is FALSE otherwise. |

The order of evaluation for complex Boolean formulas depends on the precedence of operators. Relational operators, all of which are of the same precedence, are higher than Boolean operators and are performed first. The precedence of Boolean operators is:

- (1) NOT
- (2) AND
- (3) OR and EQUIV

Parentheses can be used in Boolean formulas to clarify or modify the order of evaluation. Parentheses must not be allowed to interfere with the precedence of relational over Boolean operators or meaningless formulas may result.

Boolean Formulas

The Boolean formulas most deeply nested in parentheses are evaluated first, the next deepest next, etc. A left-to-right order of evaluation applies after the application of parentheses and precedence rules. During the left-to-right evaluation, the Boolean formula's value is completely specified as soon as the evaluation of a comparison conclusively determines a value for the entire formula. In a set of comparisons connected by ANDs, the value FALSE is specified for the entire statement as soon as any comparison specifying FALSE is evaluated; and similarly, a set of comparisons connected by ORs specifies TRUE as soon as any comparison specifying TRUE is evaluated. Consider the evaluation of the complex formula

(BOOL1 OR BOOL2) AND (BOOL3 OR BOOL4) OR BOOL5

If both BOOL1 and BOOL3 give the value TRUE, they will be the only two relations tested to determine the formula's value to be TRUE. If BOOL1 and BOOL2 both give the value FALSE, they will be the only two relations tested to determine the value of the two parenthetical formulas linked by the AND operator to be FALSE; hence only BOOL5 remains to be tested to determine the condition of the entire formula.

Examples (Complex Boolean Formulas)

For the following examples, c_i refers to a Boolean formula

$$(1) \quad c_1 \text{ OR } c_2 \text{ AND } c_3 \text{ OR } c_4$$

The AND is binding c_2 and c_3 while the ORs are separating the statement into three general conditions. Any one of the following three conditions makes the formula TRUE:

- (a) c_1 is TRUE
- (b) c_2 and c_3 are TRUE
- (c) c_4 is TRUE

$$(2) \quad c_1 \text{ AND } c_2 \text{ OR } c_3 \text{ AND } c_4$$

There are two general conditions that make the formula TRUE:

- (a) c_1 and c_2 are TRUE
- (b) c_3 and c_4 are TRUE

(3) c_1 AND (c_2 OR c_3) AND c_4

Parentheses group two conditions and thereby change the logic of the statement. For the formula to be TRUE, the following three conditions must all be TRUE:

- (a) c_1 is TRUE
- (b) c_2 or c_3 or both are TRUE
- (c) c_4 is TRUE

Operator Precedence

The operators at the top of the following list have greatest precedence; they will be performed before operators lower on the list when all appear at the same level of parenthesized nexting. Operators with equal precedence are on the same line. If two operators of equal precedence appear in an expression, the leftmost operation is performed first.

| | |
|------------|-------------------------------|
| Numeric | unary - ** *, / +, - |
| Logical | LSH, RSH LAND LOR, LXOR |
| Relational | EQ, NQ, GR, GQ, LS, LQ |
| Boolean | NOT AND OR, EQUIV |

Example

The formula

$A*B/C**2$ RSH 3 GR D OR E AND F LXOR G NQ 0

is equivalent to

$((((A*B)/(C**2)) \text{ RSH } 3) \text{ GR } D) \text{ OR } (E \text{ AND } ((F \text{ LXOR } G) \text{ NQ } 0))$

2 3 1 4 5 9 8 6 7

with the order of operations indicated underneath each operator. This is a Boolean formula since the last operation performed is the Boolean OR.

A simple assignment statement is used to specify that a formula is to be evaluated and the resulting value assigned to a variable. Multiple, nonscalar, and exchange assignment statements can be decomposed to simple assignment statements; the rules given here apply to them as well unless otherwise stated.

Format

variable = formula

The variable on the left side of the equals sign must be a scalar variable in a simple assignment statement and may also appear in the formula on the right. In this case, the old value of the variable is used in evaluating the formula. In what follows, the variable is referred to as the left term and the value of the formula as the right term.

The type of variable designated by the left term defines the type of assignment statement and may be any data type in the language. With the exception that textual formulas may be assigned only to textual variables, the right term need not agree in type with the left and may itself contain data of different types.

In evaluation of a formula containing operands of different types, the intermediate type of their combination will be that which gives the greatest precision. In the absence of a floating-point capability, fixed point is assumed to give the greatest precision and thus is the most dominant, followed in turn by integer and then Boolean. So long as floating point is not included, no conversion will be performed during formula evaluation. (Integer and Boolean data can be considered to be fixed-point data with 0 fractional bits if fixed point is dominant, and Boolean data can be considered to be of integer type if integer is dominant.) If floating point is included, however, it dominates fixed point and appropriate fixed-to-floating conversions are performed.

The resulting type of the right term is determined by the last operation performed in the formula evaluation. If this type is logical, the value of the formula, bit for bit, will be assigned to the left term without conversion, regardless of the type of the left term. If the result is longer than the left term, the excess bits will be truncated from the high-order end of the result before assignment takes place. If the result is shorter, it will be stored in the left term right-adjusted, with zeros filling the excess high-order bits.

Assignment

The result of the evaluation will be automatically converted to the type of the left term according to the following rules.

Assigning to Fixed-Point Variables

- (1) If the right term is fixed point, it is adjusted by shifting so that its binary point is aligned with that of the fixed-point variable to which it is being assigned. A direct transfer of data then takes place. If the left and right terms have identical binary points, no shift will be generated. Truncation of the fractional part may result when the scaling readjustment is done. No truncation or overflow beyond that resulting from the scaling readjustment will take place if the integer part or fractional part of the right term is greater than the corresponding part of the left term.
- (2) If the right term is an integer or Boolean quantity, it is treated exactly as a fixed-point quantity whose binary point is immediately to the right of its least significant bit.
- (3) If the right term is a floating-point quantity, it is converted to fixed point of same scaling as that of the left term.

Assigning to Integer Variables

- (1) If the right term is a fixed-point quantity, it is adjusted by shifting so that its binary point is positioned immediately to the right of its least significant bit. An equal transfer of data then takes place. Any fractional part will be truncated by this operation. Overflow may result in the integer part if that part is larger than the amount of space allocated to the integer item.
- (2) If the right term is an integer or Boolean quantity, the result is an equal transfer of data.
- (3) If the right term is a floating-point quantity, it is converted to an integer quantity.

Assigning to Boolean Variables

- (1) If the right term is a fixed-point, integer, or floating-point quantity and if all bits are 0, then 0 (FALSE) will be assigned to the left term. Otherwise, the value 1 (TRUE) will be assigned.
- (2) If the right term is a Boolean quantity, an equal transfer of data takes place.

Assigning to Textual Variables

- (1) Fixed-point, integer, Boolean, and floating-point quantities may not be assigned to textual variables.
- (2) If the right term is textual and is the same size as the variable being set, an equal transfer of data takes place. If it has more characters than the variable, the rightmost characters are lost; and if it has fewer characters, the value is assigned left-justified within the variable and excess character positions are set to blank.

Assigning to Floating-Point Variables

- (1) If the right term is floating point of the same precision as that of the left term, a direct transfer takes place. If the right term is double precision and the left term single precision, the excess word is truncated; if the left term is double and the right single, the excess word is set to 0.
- (2) If the right term is integer, fixed point, or Boolean, it is converted to floating point with the same precision as the left term

Examples

- (1) DECLARE FIXED 24 5, DIST, SPEED, TIME
:
:
:
DIST = TIME * SPEED / 3600

DIST is computed in fixed point. The 3600, although an integer constant, will be interpreted as fixed point since it is being divided into a fixed-point operand.

Assignment

(2) DECLARE INTEGER, ALPHA, BETA, GAMMA

:

ALPHA = (BETA NQ 0) * GAMMA

ALPHA is set to 0 or GAMMA depending on the value of the Boolean formula (which is interpreted as an integer, 0 or 1, for purposes of numeric computation).

(3) DECLARE BOOLEAN, ANSWER (3), REPLY

:

ANSWER (0) = FALSE

:

ANSWER (1) = BETA GR ALPHA OR ANSWER (0)

:

ANSWER (2) = ANSWER (0) EQUIV ANSWER (1)

:

REPLY = ANSWER (2)

Elements within the Boolean array ANSWER are set to Boolean formulas, and a Boolean assignment is made between an element in the array ANSWER and the Boolean item REPLY.

(4) DECLARE INTEGER, IOTA
DECLARE FIXED, EPS 24 10

:

IOTA = B'01010'

EPS = IOTA LOR B'00001'

IOTA is set to B'01010' and EPS to B'01011'. Because EPS is assigned to a logical formula, no conversion or rescaling takes place even though different types are involved -- only a transfer of bits.

The TEMP attribute is used to prevent the compiler from rescaling items, that is, when it is desired to employ an item for temporary storage of fixed-point values having many different scalings. Thus, whenever a fixed-point item previously declared with TEMP is used on the left side of an assignment statement, a logical assignment occurs: a transfer of bits takes place without regard to the scaling of the quantity being assigned. The TEMP variable will assume the same word size and number of fractional bits (the same scaling) as the formula on the right, and it will keep this scaling until the TEMP variable again appears on the left of an assignment statement as determined by a beginning-to-end scan of the program. Whenever the TEMP variable appears on the right of an assignment, it will be assumed to have the currently effective scaling attributes. This may result in some conflicts, the resolution of some of which is shown in the example.

Example

```
DECLARE FIXED, R TEMP
```

```
:
```

```
ALFA. R = A + B
```

```
W = R / (R + C)
```

```
R = W * X
```

```
Z = R / (R - Z)
```

```
BETA. L = J + R
```

When code is generated by the compiler, R will assume the temporary attributes of the expression $A + B$. Each time an assignment is made to R as the program is scanned from beginning to end, R will take on local attributes that will be effective until the next assignment to R. When a new statement label is encountered after an assignment to R has been made, the local attributes currently in effect for R may not be valid; thus a diagnostic message will be issued by the compiler which includes a pointer to the statement from which the currently effective local attributes have been derived. For this example, a diagnostic message will be printed for the statement labeled BETA and indicating that the local attributes of R have been determined at statement ALFA+2.

Scaling Operator

The scaling operator is used to override intermediate scalings that would normally result from the application of scaling algorithms during evaluation of fixed-point formulas. One form enables the total number of bits and the number of fractional bits to be declared for intermediate fixed-point results.

Format

$.S(\{n_1, \} n_2)$

.S is the scaling operator and the n's are integer constants. The optional n_1 indicates the total number of bits with which the intermediate result is to be maintained -- if absent, a full word is assumed. n_2 is the number of fractional bits to be maintained in the intermediate result.

The scaling operator is written immediately to the right of the fixed-point operand (variable or formula) which is to be scaled. The value of the operand will be shifted if necessary to satisfy the indicated scalings. If the operand is a formula containing operators, it must be surrounded by parentheses.

Example

$A = (B * C) .S(10, 7) + D$

The scaling operator indicates that regardless of the previous declarations as to the size and precision of items B and C, a 10-bit total length with 7 fractional bits is sufficient for their product in this formula. This assignment is equivalent to the following:

```
DECLARE FIXED, IRESLT 10 7
:
:
IRESLT = B*C
A = IRESLT+D
```

Thus the effect of the scaling operator on its operand is the same as what would occur if the operand were assigned to a variable having the same bit declarations as the arguments of the scaling operator.

Another form of the scaling operator, `.S` by itself, is used to indicate that the intermediate result is to be maintained in extended precision at a small cost in generated object code.

Example

```
A = (B * C) .S + (D * E) .S
```

Although A, B, C, D, and E are items each occupying a single computer word or less, the intermediate products `B * C` and `D * E` will be retained in the double-precision form in which they are obtained from the computer's multiply instruction, and the products will also be summed using double precision. The assignment of the result to A will involve a single-precision store, however. Note that `.S` is meaningful for multiplication results only to the extent of indicating how summing is to occur.

A third form of the scaling operator, `.S` followed by a statement label enclosed in parentheses, is used to override the automatic scalings determined by the compiler for TEMP variables. The scaling used will be that determined at the statement label which is the argument of the scaling operator.

Example

```
DECLARE FIXED, T1 TEMP, V1 15, V2 0, V3 0
:
:
L1. T1 = V1
:
:
GOTO L2
:
:
V2 = T1/3.14159
:
:
T1 = V2
:
:
L2. V3 = T1.S(L1) * V2
```

Scaling Operator

T1 is first used to store V1 temporarily. When T1 is used in the formula assigned to V2, it is considered to have the same scaling as V1 (that is, 15 fractional bits) because of its local scaling. In the last statement, T1 is again used with the scaling determined by the assignment statement labeled L1 because the .S(L1) overrides the new local scaling of 0 fractional bits determined by the scaling of V2.

A multiple assignment statement is used to set several variables via one statement. It performs the same operations as several simple assignment statements but requires less writing.

Format

$$\text{variable } \{, \text{variable} \}_1^{\infty} = \text{formula } \{, \text{formula} \}_0^{\infty}$$

The number of variables must not be less than the number of formulas.

The formulas are evaluated and the results assigned to each formula's corresponding variable on the left. If there are fewer formulas on the right than variables on the left, the unmatched variables are set to the value of the last formula on the right; see Example (2).

Multiple assignment can also be accomplished by use of a previously declared group name for the variable on the left; all declared items in the group will be multiply assigned in the order of their declaration. See Example (3). The group name should not be used on the right of an assignment statement.

Examples

(1) A, B, C = 4, Q/R, ABS(Y)

This multiple assignment is equivalent to the following three simple assignment statements:

A = 4
B = Q/R
C = ABS(Y)

(2) A, B, C, D = 5, 6

This is equivalent to:

A = 5
B = 6
C = 6
D = 6

Multiple Assignment

(3) GR5. DECLARE FIXED, ALPHA, BRAVO, ECHO

:

GR5 = 0.0

:

GR5 = A+B, 7.0, ALPHA**2

In the first assignment statement, all items in GR5 are set to 0.
In the second, ALPHA is set to A+B, BRAVO to 7.0, and ECHO
to ALPHA**2.

Nonscalar assignment statements, that is, those containing nonscalar items as operands, are used to replace, in a more succinct form, several equivalent simple assignment statements.

Format

variable = formula

The variable on the left side of the equals sign must be a nonscalar variable and may also appear in the formula on the right. If the formula on the right is scalar (one in which nonscalar items do not appear), then it is evaluated once and assigned to all elements of the variable on the right as indicated in Example (1). If the formula on the right is nonscalar, it is evaluated for all possible values of the nonscalar subscripts (asterisks) it contains and the results are stored in the corresponding positions of the left term. For this to be accomplished, each nonscalar item in the right term must agree with the left term according to the following two rules:

- (1) It must have the same number of asterisk subscripts as the left term.
- (2) Each asterisk on the right must represent the same size dimension as the corresponding asterisk of the left term.

Illustrations of the violations of these two rules are provided in Example (2) below.

A special nonscalar assignment operation is provided for accomplishing the linear algebra type of matrix multiplication where

$$A = B \cdot C$$

such that for each element A_{ij} of matrix A

$$A_{ij} = \sum_{k=1}^n B_{ik} \cdot C_{kj}$$

Nonscalar Assignment

Format

$$m_1 = m_2 /* / m_3$$

where the m's are two-dimensional array identifiers declared such that the number of columns of m_2 equals the number of rows of m_3 and the dimensions of m_1 are the same as the row dimension of m_2 and the column dimension of m_3 .

This operation, the only nonscalar operation accomplished without using asterisk subscripts, is illustrated in Example (7). The `/* /` operator may not be used in any type of formula other than the one described. n-element one-dimensional arrays are considered to be $n \times 1$ matrices for purposes of matrix multiplication.

Examples

(1) DECLARE INTEGER, MAT(3, 3)

⋮

MAT(*, *) = MAT(0, 0)

⋮

MAT(*, 2) = 0

In the first nonscalar assignment, all elements of array MAT are set to the value of the first element of MAT. In the second assignment, the last column of MAT is set to all zeros.

(2) DECLARE FIXED 24 0, X(3, 4, 5), Y(3, 5), Z(4, 3)

⋮

X(*, 2, *) = Y(*, *)

⋮

X(*, *, K-2) = Z(*, *) (illegal)

⋮

X(*, 0, 0) = Y(*, *) (illegal)

The first assignment sets the cross-sectional matrix formed by column 2 in all planes of X equal to the matrix Y. The second assignment is illegal because it violates Rule (2): corresponding asterisks do not represent the same size dimensions. The third assignment is illegal because it violates Rule (1): the number of asterisks on both sides do not agree.

(3) `SINX (*) = .SIN(X(*))`

Each element of vector SINX is set to the value of the function .SIN applied to corresponding elements of vector X. .SIN will be called separately for each element of X and thus will receive a scalar value for an argument.

(4) `DECLARE FIXED (3,2), A,B,C`

`:`
`:`

`A(*,*) = B(*,*) + C(*,*)`

Matrices A, B, C are declared through a factored declaration which sets the elements to fixed point and the size of all three matrices to 3x2. Every element of B is added to the corresponding element in C and stored in the corresponding element of A. The nonscalar assignment is equivalent to the following six scalar assignments:

`A(0,0) = B(0,0) + C(0,0)`

`A(1,0) = B(1,0) + C(1,0)`

`A(2,0) = B(2,0) + C(2,0)`

`A(0,1) = B(0,1) + C(0,1)`

`A(1,1) = B(1,1) + C(1,1)`

`A(2,1) = B(2,1) + C(2,1)`

(5) `A(*,*,5) = B(*,*) * 3.0`

All elements of matrix B are multiplied by 3.0 and the results are stored in the corresponding positions of plane 5 in three-dimensional matrix A. The number of rows and columns in a plane of A must be the same as the number of rows and columns of B.

Nonscalar Assignment

(6) $T(*) = X(*)/5 + Y(*) * W * U - Z(*) ** 2$

This complex nonscalar assignment involves four different nonscalar variables. The compiler will optimize by computing scalar subexpressions such as $W * U$ only once and saving the result for use during the evaluation of the formula for each possible value of the subscripts.

(7) DECLARE INTEGER, A(M, T), B(M, N), C(N, T)

:

A = B /*/ C

This is an example of nonscalar assignment for performing the linear algebra type of matrix multiplication.

An exchange statement is used to interchange the values of scalar or non-scalar variables via a single statement. With nonscalar variables, it can be employed to interchange rows or columns of a matrix.

Format

$$\text{variable } \{, \text{variable} \}_0^{\infty} == \text{variable } \{, \text{variable} \}_0^{\infty}$$

The number of variables on the left of the exchange operator, $==$, must equal the number of variables on the right. Each variable on the left will be exchanged with its corresponding variable on the right. The rules of assignment pertain in both directions.

Examples

Assuming that the following data declaration has been made:

```
DECLARE INTEGER, A(10), B(10), C(4, 4), D(5, 5),
              U, V, W, X
```

(1) $A(2) == B(4)$

The third element of array A and the fifth element of array B are interchanged.

(2) $C(2, *) == C(3, *)$

The third and fourth rows of array C are interchanged.

(3) $D(*, 1) == D(*, 3)$

The second and fourth columns of D are interchanged.

(4) $U, V == W, X$

Simple variables are exchanged. This single exchange statement is equivalent to the following six simple assignment statements:

Exchange Assignment

TEMP = U

U = W

W = TEMP

TEMP = V

V = X

X = TEMP

PROGRAM CONTROL

The statements in a CLASP program are executed in the sequence in which they appear except as this sequence is altered, either absolutely or under specified conditions, by control statements. Any statement may be labeled to permit it to be referenced by other statements. The simple GOTO statement may be used to transfer control to any labeled statement, and the switched GOTO statement provides for transfer of control to any one of a group of labeled statements depending on the value of an index. Conditional statements may be used to transfer control dependent on the result of a Boolean formula. Loop statements are provided for causing the same sequence of code to be executed a specified number of times, with an index modified each time by a specified amount. The inhibit/enable statements LOCK and UNLOCK are provided for inhibiting or activating hardware functions such as interrupts and for preventing the use of registers by the object code in designated program areas. The chronic statement ON allows the sequence of control to be altered when some enabling condition, such as a hardware interrupt, occurs. Finally, the STOP statement may be used to cause a halt in the execution of the object program

Statement Labels

Statement labels are used to name statements, allowing such labeled statements to be referenced by other statements. The statement label, which consists of an identifier immediately followed by a period, may be placed at the beginning of any executable statement. References to a labeled statement may be made both before and after the label is defined in the sequence of statements. Statements which are not referenced by other statements need not be labeled.

Example

```
STAGE. ABLE = Y*Z
```

STAGE is the statement label. If transferred to, ABLE will be assigned the value $Y*Z$ and the next sequential statement will be executed.

The GOTO statement is used to transfer program control to the statement whose label is used with the GOTO.

Format

GOTO identifier

The identifier must be a statement label defined in the program. Note that the identifier is used without the period which is required when labeling statements.

Example

```
GOTO FOX
:
:
FOX.  A = B+C
```

Control is transferred to statement FOX by the GOTO. All statements in between are bypassed.

Switched GOTO Statements

The switched GOTO is used to transfer control to one of many points in the program based on the value of an associated index.

Format

GOTO ({identifier} {', identifier'}₀[∞]) integer-formula

The identifiers in the parentheses (the switch list) must be statement labels, and the integer formula (the index) must be of the same form as that required for array subscripts (that is, no more complicated than $v * c_1 + c_2$ or $v / c_1 + c_2$).

When the switched GOTO is executed, control is transferred to the statement whose position in the switch list corresponds to the current value of the index: if the value is 0, transfer is made to the first statement in the list; if 1, to the second statement; etc. Transfer of control does not occur if the index value:

- (1) corresponds to an empty position (without a statement label) in the switch list,
- (2) is a negative number, or
- (3) is greater than the number of commas in the switch list.

With no transfer of control, the statement following the GOTO is executed. If the switch invokes a close, the statement following the GOTO is executed when the close exits.

Example

```
GOTO (A,, B, C, D) ABLE
Q. W = X + Y
```

Control will be transferred to A if ABLE = 0; to Q, the statement following the GOTO, if ABLE = 1; to B if ABLE = 2; etc.

An alternate form of the switched GOTO is written with an asterisk in the last position of the switch list to speed up the GOTO and save object code.

Format

GOTO ({identifier} {[∞]₁, identifier[∞]₁, *}) integer-formula

The asterisk serves to inhibit limit comparisons on the index; it is used when it is assumed that the index always corresponds to a position on the switch list. If the index is negative or is greater than or equal to the number of commas, unpredictable results will occur using this form.

Example

GOTO (B1, B2, B3, *) J/2

If this GOTO is activated, the branch will execute normally for $J = 0, 1$ (control will be transferred to statement B1), for $J = 2, 3$ (transfer to statement B2), or for $J = 4, 5$ (transfer to statement B3). The branch will be undefined for all other values of J .

Conditional Statements

The conditional statement is used to transfer control or execute a section of code based upon the evaluation of a Boolean formula. The general format is the primitive IF followed by a Boolean formula, a THEN statement group, an optional ELSE statement group, and the primitive END or ENDALL. If the Boolean result is TRUE, the THEN statement group is executed and the ELSE group, if present, is bypassed. If it is FALSE, the THEN group is bypassed and the ELSE group, if present, is executed. After the THEN or ELSE statement group is executed, as appropriate, control is transferred to the statement following the END or ENDALL unless another transfer of control has been executed. Conditional statements may be nested; hence additional conditional statements may appear in both the THEN and the ELSE statement groups.

Format (with an ELSE group)

IF Boolean-formula

THEN statement

{statement}₀[∞]

ELSE statement

{statement}₀[∞] END
ENDALL

The THEN group is delimited by THEN and ELSE and the ELSE group is delimited by ELSE and END or ENDALL.

Format (without an ELSE group)

IF Boolean-formula

THEN statement

{statement}₀[∞] END
ENDALL

Since there is no ELSE statement group in this form, if the Boolean result is FALSE nothing will be executed and control will be transferred to the statement immediately following the END or ENDALL.

Each line of the two forms is a separate statement. These statements should be written according to the normal rules (page 14) with the exceptions that:

- (1) The IF and THEN statements may be written on the same line. If this is done, a statement terminator need not be used between the Boolean formula and the THEN.
- (2) The last statement of the THEN group and the ELSE statement may be written on the same line. If this is done, a statement terminator need not be used between the two statements.
- (3) The END or ENDALL may be appended to the last statement in the conditional statement group or may be written as a separate statement. It may be labeled if written as a separate statement. If labeled and if transferred to, control is transferred to the statement immediately following.

ENDALL is used to close out nested conditional statements (and also nested loop statements as defined in the next section); whenever an ENDALL is encountered, every open conditional and loop statement is closed. Use of the ENDALL avoids a string of ENDS at the end of nested statements.

Examples

- (1) IF A GR B AND A LS C THEN GOTO STACK
ELSE IF A EQ D THEN GOTO (P0, P1, P2) I
ENDALL
P3. X = W + Y

This is a nested conditional statement. If A is greater than B and less than C, control is transferred to statement STACK. If not, and if A equals D, the switched GOTO is executed, but if A does not equal D, control is passed to the next sequential statement, P3.

- (2) IF A
THEN C(*) = B(*)
X = Y - 3.0
END
I = J - K

Conditional Statements

This is a conditional with no ELSE group. If A is TRUE (non-zero), the vector and scalar assignment statements will be executed, followed by the statement after the END. IF A is FALSE, the two assignments will be bypassed and the statement following the END will be executed.

```
(3) IF ABLE LS BAKER  
    THEN C = 5.0 * D  
    GOTO EVAL  
    ELSE C = 2.5 * D END  
    GO. A = B + C
```

When ABLE is less than BAKER, the THEN group is executed such that C is set to 5.0 times D and control passes to the statement EVAL. Otherwise, C is set to 2.5 times D and control passes to the next statement, GO.

Loop statements are used to cause a segment of coding to be repetitively executed one or more times.

Format

```
FOR v = i {BY s} TO t
```

```
{statement}0∞ END  
ENDALL
```

v is the loop variable, i is the value to which it is initially set, s is the optional step size (assumed to be 1 if omitted), and t is the limit. v must be of integer type and i, s, and t may be integer variables or constants. Note that each line is written as a separate statement, except that the END or ENDALL may be appended to the last statement in the loop.

When a loop is entered, all statements up to the END or ENDALL are executed; then v is incremented by s and compared with t. Control is returned to the first statement following the primitive FOR and the process is repeated until the limit comparison shows v to be greater than t (for positive s) or less than t (for negative s). At this point, the loop is exited and control is passed to the next statement after the END or ENDALL.

The loop statement provides a shorthand method of accomplishing tasks that can also be done with conditional statements; the following examples illustrate equivalent coding:

Loop Statement

```
FOR I = 1 BY 2 TO N  
Statement-1  
Statement-2  
Statement-3  
END
```

Conditional Statement

```
I = 1  
S1. Statement-1  
Statement-2  
Statement-3  
I = I + 2  
IF I LQ N THEN GOTO S1  
END
```

Loops may occur within other loops; each level of iteration after the first is part of the previous loop. Inner loops are always completed before outer loops. The resulting loop structure may have several ENDS in a row. In general, each END generates an increment and test. If an END is labeled

Loop Statements

(which requires that it be written as a separate statement) and transferred to, incrementing and testing will begin with the loop variable corresponding to that END. ENDALL is used, as with conditional statements, to close out all open loops and conditional statements with a single statement. If an ENDALL is labeled and transferred to, the innermost loop variable will be incremented and tested first, then the next innermost, and so on.

The loop variables for the individual loops within a series of nested loops must not be the same. If the step size or limit are variables, they will be reevaluated for each iteration of the loop; hence their values may be altered in the loop if desired. The loop variables may also be altered in the loop. When the step size is coded as a variable, its sign must be tested to determine whether the loop variable is to be tested for being greater or less than the limit; hence execution time and object code are saved by coding constant values for the step size.

Control may not be transferred from outside a loop to statements within its range. Once in a loop, however, control may be passed to statements outside of the loop. Once a loop is exited, either by normal termination of iterations or by transferring out of its range, the value of the loop variable is undefined.

Examples

```
(1)  FOR J = M BY N TO P
      A(J) = A(J) + B(J)
      IF A(J) EQ 0.0 THEN GOTO LPEND END
      B(J) = B(J - 5)
      LPEND.  END
```

J is initialized with M, the statements are executed, and when the statement LPEND is executed, J is incremented by N and tested. Note that since the step size, N, is a variable, its sign must first be tested to determine whether J is to be tested for being greater or less than the limit, P.

```
(2)  FOR I = 0 TO N
      FOR J = 0 TO M
        B(I, J) = A(J, I)
        A(J, I) = 0 ENDALL
```

This is an example of nested loops. The J loop will be executed $M+1$ times for every iteration of the I loop. A total of $(N+1) \times (M+1)$ iterations will occur.

Inhibit/Enable Statements

The inhibit/enable statements LOCK and UNLOCK have three uses:

- (1) To inhibit or enable an interrupt previously declared via a chronic statement
- (2) To protect an area of memory from or make it available for writing (for object computers having a memory-protect capability)
- (3) To reserve exclusive use of a register or registers for the programmer.

Format

```
LOCK    interrupt-name
UNLOCK  storage-location TO storage-location
        register-name
```

$$\left. \begin{array}{l} , \text{interrupt-name} \\ , \text{storage-location TO storage-location} \\ , \text{register-name} \end{array} \right\}^{\infty}_0$$

The interrupt and register names are implementation-dependent hardware codes. The storage locations may be absolute core addresses indicated by constants or the location of items indicated by identifiers.

The first use of inhibit/enable statements is illustrated in Example (1) and the second in Examples (2) and (3). With regard to the third application, the LOCK statement tells the compiler that it can no longer use the indicated registers in the compiled code except where the programmer references them. If compilation cannot continue without a given register, the compiler will save the programmer-specified value and restore it when through using the register. A diagnostic will be issued if this has to be done. This third use of the inhibit/enable statement pair is illustrated in Example (4).

Examples

```
(1) LOCK OVERFLOW, CLOCK
    :
    :
    UNLOCK CLOCK
```

OVERFLOW and CLOCK must have been declared in an ON statement.

(2) LOCK DATA TO START

The memory area from the location of DATA to the location of START is protected.

(3) UNLOCK 0 TO 63

The memory area from location 0 to location 63 is made available.

(4) DECLARE HARDWARE I, REG1 = 2

:

LOCK 2

REG1 = X

:

UNLOCK 2

By means of the HARDWARE declaration, register 2 is given the the name REG1. Hardware code 2 is put under programmer control by the LOCK statement and will not be used for holding values in the following statements except where use of REG1 is specifically indicated by the programmer. Register 2 is returned to compiler control by the UNLOCK statement.

Chronic Statements

The chronic statement, ON, is used to indicate the statements to be executed upon the occurrence of a specified enabling condition, usually a hardware interrupt. Chronic statements are not part of the normal sequence of program execution.

Format

ON interrupt-name

{statement}₁[∞]

EXIT

The interrupt name is an implementation-dependent hardware code for the particular interrupt.

When the interrupt occurs, execution of the current task is automatically suspended and control is passed to the first statement following the ON. When the EXIT statement is executed, control is returned to the task which was interrupted.

The programmer has complete control over the conditions under which a chronic statement is executed. Statements may be organized so that a low-priority task does not interrupt a high-priority task or process by using inhibit/enable statements as indicated in the previous section.

Example

Assuming that 10MSI and 5MSI are the hardware codes for 10- and 5-millisecond interrupts:

```
ON 10MSI
LOCK 5MSI
CNT10 = CNT10+1
.UPDATE
.NAVIG
UNLOCK 5MSI
EXIT
```

When the 10-millisecond interrupt occurs, the currently executing task will be interrupted, the lower priority 5-millisecond

Chronic Statements

interrupt will be inhibited, and the update and navigation routines will be called. The 5-millisecond interrupt will be enabled following completion of the navigation routine, and control will be returned to the task that was interrupted.

STOP Statements

A STOP statement is used to cause a computer halt.

Format

STOP {identifier}

The optional identifier is a statement label without a period.

After a STOP is executed, a computer restart operation will result in a transfer of control to the next statement unless an identifier has been specified, in which case a transfer is made to the statement named.

SUBPROGRAMS

Subprograms provide the capability to specify particular computations in one place in a CLASP program and to call for their performance from many places. Three types of subprograms may be defined: procedures, functions, and closes. A procedure operates on data called input parameters, performing calculations with them and producing results which are stored in output parameters. Data may be declared locally in a procedure independent of data in the main program or another procedure. A function has input parameters but no output parameters. Instead, a single result is calculated which is passed via the name of the function, allowing it to be used as an operand of a formula. A close has neither input nor output parameters. It cannot have local data declarations; the data it manipulates must be declared in the main program.

In addition to the subprograms that may be defined by the programmer, several library subprograms are available for performing commonly used, simple functions such as rounding and limiting.

Procedure Declarations

A procedure declaration is used to describe the input and output parameters of a procedure, data local to the procedure, and the computations to be performed when the procedure is called. The procedure declaration is divided into three parts: the procedure description, the heading, and the body.

Format (Procedure Description)

```
PROC .identifier {(input-parameter {, input parameter }0∞)  
                  {=output-parameter {, output-parameter }0∞) } {INLINE }
```

The identifier which follows the primitive PROC is the procedure name and must be preceded by a period. The optional input and output parameters with which the procedure is declared are enclosed in parentheses and are called dummy parameters. Each input parameter may be a simple item name, an array name, or a close name; a close name must be preceded by a period. The equals sign separates input from output parameters and is omitted if no output parameters are specified. Each output parameter may be a simple item name, an array name, or a statement label; a statement label must be followed by a period. The optional attribute INLINE causes the procedure to be inserted directly in-line whenever it is called; any dummy parameters are directly replaced with actual parameters before the code is generated. If INLINE is not specified, procedures are in closed form and appear only once in the code. When they are called, execution time must be spent transferring back and forth and passing parameters. This is not required with in-line procedures.

The procedure heading contains data declarations of all input and output parameters except close names and statement labels; these are declared implicitly by the periods. Any arrays used as dummy parameters may be declared with variable dimensions if the dimensions are described via input parameters. Variables local to the procedure are also declared in the heading. These local variables cannot be referenced in the main program or in any other procedure.

The procedure body is a list of CLASP statements; it does not include procedure or function declarations but may call them. These statements use

the dummy input parameters, which are converted to the actual input parameters when the procedure is called. Therefore, the actual input parameters must agree by declaration with the declarations of the dummy input parameters. Variables declared in the main program may be referenced in the procedure body; such variables are called global variables. If a dummy parameter or local variable has the same name as a global variable, the compiler will assume that the parameter or local variable, not the global variable, is being referenced when the name is used in the procedure.

The last statement of the procedure body is always EXIT. When it is executed, control returns to the statement following the procedure call.

Examples

```
(1) PROC .CMPUTE (X, Y, .CLS = RES, ERR.)
    DECLARE FIXED, X 24 10, Y 22 0, RES (10) 24 10
    DECLARE FIXED, L1 24 10, L2 24 0, K INTEGER
    IF X LS 0 THEN GOTO ERR END
    L1 = X
    :
    :
    FOR K = 0 BY 1 TO 9
    RES(K) = K * L1 / Y - X END
    EXIT
```

CMPUTE is declared as a procedure having as input parameters X and Y (which are fixed point) and CLS (which is a close). Its output parameters are RES (a fixed-point array) and ERR (a label). L1, L2, and K are declared locally. In the body of the procedure, X is tested and if less than 0, control is passed to statement ERR, which is outside of the procedure. This is an abnormal exit. Later on in the body, output array RES is given values in the FOR loop. The procedure is then exited.

```
(2) PROC .BOOST (A, B, C)
    DECLARE I, A, B, C
    :
    :
    EXIT
```

This is an example of a procedure with no output parameters.

Procedure Declarations

```
(3) PROC    GUIDE
      :
      :
      EXIT
```

Procedure GUIDE is declared with no formal parameters.

A procedure is called by using its name in a statement with actual parameters substituted for the dummy parameters with which it was declared.

Format

$$. \text{procedure-name} \{ \{ \text{input-parameter } \{, \text{input-parameter} \}_0^\infty \} \{ \text{=output-parameter } \{, \text{output-parameter} \}_0^\infty \} \}$$

The procedure name is the unique identifier with which the procedure was declared; it must be preceded by a period. The actual input parameters may be constants, variables, array names, formulas, or close names (preceded by a period). The equals sign separates the input parameters from the output parameters and is omitted if no output parameters are specified. The actual output parameters may be variables, array names, or statement labels (followed by a period).

A reference made to a dummy parameter in the procedure body generates a reference to the actual parameter whose position in the procedure call parameter list corresponds to the position of the dummy parameter in the description portion of the procedure declaration. The table below indicates how the dummy parameters and the actual parameters must agree, and the last column indicates how data are transmitted to the procedure.

PROCEDURE PARAMETERS

| Type | Dummy Parameter | Actual Parameter | Called By |
|--------|--|--|-------------------------------|
| Input | Nontextual item name Array name Textual item name Close name | Nontextual formula Array name Textual formula Close name | Value Name Name Name |
| Output | Nontextual item name Array name Textual item name Statement label | Nontextual variable Array name Textual variable Statement label | Value Name Name Name |

Procedure Calls

Arrays and textual items are called by name. The data are never moved; only the data addresses are passed through the parameter. These parameters are operated on as if the actual parameter names had been substituted for the dummy parameter names in the procedure body. The procedure does not know how the actual parameter is constructed when the dummy parameter is called. Thus, if the dummy parameter is a two-dimensional array, for example, the actual parameter must be an array with the same dimensions.

Nontextual items are called by value. The procedure is executed as if the values of the actual input parameter formulas were assigned to the dummy input parameter items before execution. The values of the dummy output parameters are assigned to the variables that are actual output parameters after execution. Consequently, there must be compatibility between dummy parameter items and actual parameter variables or formulas identical to that required for assignment statements.

The actual input parameters are evaluated in a left-to-right sequence. Following evaluation of the input parameters, any indices of the actual output parameters are evaluated from left to right. Values then are assigned to dummy input parameters before execution of the procedure body. Upon return from the procedure, the dummy output parameters are assigned to the corresponding actual output variables.

A procedure exits normally by execution of the EXIT statement; all called-by-value output parameters are set and control is returned to the statement following the procedure call. Statement names appearing in dummy output parameter lists are called alternate exits. If a GOTO or STOP that references a dummy output parameter is executed in the procedure, control is returned to the statement label in the actual output parameter. If control is passed to the main program through a GOTO or STOP statement, the final assignment process is bypassed and the actual called-by-value output parameters are not changed. In addition, it is possible that main program loop variables that were active at the time of calling the procedure will not have their correct values; this can happen if the procedure itself activates any loop variables. If so, the main program loop variables will be saved but not restored if the procedure is exited by a direct transfer to the main program. If a procedure is exited normally through EXIT, all loop variables are restored.

Examples

(1) .LOCATE (LAT(A), LONG(A) = SECTOR, RANGE)

This is a two-input, two-output procedure call requesting a procedure identified as LOCATE to operate. Assuming that LOCATE had been declared as

```
PROC .LOCATE (LT,LG = S, R)
```

LAT(A) and LONG(A) are actual input parameters corresponding to the dummy parameters LT and LG, respectively. Similarly, SECTOR and RANGE are the actual output parameters corresponding to dummy output parameters S and R.

(2) .DUMP

This is a no-input, no-output procedure call requesting procedure DUMP to operate.

(3) .DATE (=DAY, MONTH, YEAR)

This is a no-input, three-output procedure call. The output from procedure DATE is placed into simple items DAY, MONTH, and YEAR.

(4) .PREDIC (SHIPID(X), MPH(X) * 1.152 - OWNMPH,
BEARING, 'VERIFY' = ERR.)

Procedure PREDIC is a four-input, one-output procedure. The output from the procedure is a statement label defined in the main program; note that it is followed by a period. When the operation of the procedure is completed, the procedure will either make a normal exit to the statement following the call or will exit to the statement labeled ERR based on some logical decision.

(5) .DELTA (X, 2 = I, ABLE(I))

Procedure DELTA is called with input parameters X and 2 and output parameters I and ABLE(I). Note that the index of array ABLE is calculated before the procedure is called; thus when ABLE(I) is set on procedure exit, the index will refer to the old value of I, not the new one calculated in the procedure.

Functions

Functions are similar to procedures; they operate on input parameters to produce a result which is returned to the calling program via the function name.

A function declaration is like a procedure declaration except that output parameters are not included and the function name, besides being declared in the procedure description, must be declared in the heading along with the dummy input parameters and local variables. In the body, the function name must be assigned a value at least once. The last value the function name is assigned before exiting is the value returned to the calling program.

A function is called by using its name with actual parameters in a context where a formula is expected; it cannot be called by using its name alone in a statement. When a function is exited, control is not returned to the next statement, as with procedures, but to that point in the calling statement which uses the function's value. In all other respects, functions operate in the same manner as procedures.

Examples

```
(1) PROC .SIN(X)
    DECLARE FIXED, X 24 21, SIN 23, Y 23
    :
    :
    SIN = Y/2
    EXIT
```

Function .SIN(X) is declared. In the heading, its input parameter, X, function name, SIN, and local variable, Y, are declared. The fact that SIN is declared in the heading indicates that .SIN(X) is a function, not a procedure. In the body SIN is given a value and the function is exited.

```
(2) IF X3 - .SIN(2*PI*F) GR 0.5 THEN GOTO LI END
```

In this IF statement, the function .SIN(X) is called with an input parameter of $2*PI*F$. When control is returned, the function's value will be subtracted from X3 and the remainder of the IF statement will be carried out.

A close is a program-dependent subprogram in that the data it manipulates must be contained in variables whose identifiers are known both to the close and to the routine calling it. Thus these data must have been declared in the main program heading or a procedure heading; not within the close. A close has no input or output parameters and is called only from within the program in which it is contained. A close may be declared within a procedure, in which case it may be called only from within that procedure.

The declaration consists of the primitive CLOSE followed by any number of statements and an EXIT statement.

Format

CLOSE .identifier

{statement}₁[∞]

EXIT

The identifier becomes the name of the close and must be preceded by a period. Execution of the EXIT statement causes a return to the statement following the close call.

If statements in the body of the close do not reference any loop variables, the location of the close is not important. If they reference loop variables which are activated outside the close, the close declaration must be within the loop statement group so that the variables will be defined. In this case the close cannot be called from outside the loop.

A close does not guarantee the preservation of loop variables that may be active when it is called, whereas a procedure does. Closes that do not activate any such variables, however, will not destroy other active loop variables. Thus, a close that activates loop variables should not be called from another loop statement group.

Close declarations may contain other close declarations. Each close may be called from any level as long as the calls are not recursive.

A close name declared in a procedure's dummy input parameter list is called by a close call or switch, and the corresponding close specified in the actual input parameter list is executed. If a close thus called activates loop

Close Declarations

variables, caution is required: since a close does not preserve the active loop variables, not only might the variables active in the procedure be destroyed, but also those active when the procedure was called. This can be avoided by using simple items in lieu of loop variables in the close.

Example

```
CLOSE .EQU17
T = T + DT
L = .LN(TAU/(TAU-T))
J1 = (TAU*L) - T
S1 = J1 - (T*L)
Q1 = (T**2/2) + (TAU*S1)
J, S, Q = VEX*J1, VEX*S1, VEX*Q1
EXIT
```

A close may be called through the program before or after it is declared. It is called by specifying the close name as a separate statement or as an identifier in a switch list.

Format

.close-name

The period precedes the close name when used alone in a statement. It is not required in a switch.

Normally a close will return control to the statement following the calling statement. However, a close may terminate abnormally by transferring control to a statement label out of its range.

Examples

(1) .EQU17

Close EQU17 is called

(2) GOTO (ABORT, CALC, COMP)I

Closes ABORT, CALC, or COMP will be called if I = 0, 1, or 2, respectively. Note that the period is not used with the close names since they appear in a switched GOTO. After the close exits, control will be transferred to the statement following the switched GOTO.

Library Subprograms

Several built-in subprograms are defined in CLASP to perform basic arithmetic and data manipulation functions. The subprogram names are considered primitives and are not prefixed with a period when called. The primitives are:

REMQUO
REM
SIGN
ABS
LIM
RND
PACK
UNPACK
UNPACX

These are all in-line subprograms; when the primitives are called the appropriate code is substituted and no transfer of control takes place.

REMQUO and PACK are procedures; the rest are all in-line functions.

Format (REMQUO)

REMQUO ($i_1, i_2 = v_1, v_2$)

where i_1 and i_2 are integer formulas and v_1 and v_2 are integer variables.

Calling REMQUO results in v_1 being set to the integer part of the division of i_1 by i_2 and v_2 set to the remainder of that division.

Example

REMQUO (5, 3 = I, J)

This results in I being set to 1 and J being set to 2.

Format (REM)

REM (i_1, i_2)

where i_1 and i_2 are integer formulas.

REM (i_1, i_2) returns the remainder of the division of i_1 by i_2 ; the result is i_1 modulo i_2 .

Format (SIGN)

SIGN(f)

where f is a numeric formula.

$$\text{SIGN}(f) = \begin{cases} +1 & \text{if } f > 0 \\ 0 & \text{if } f = 0 \\ -1 & \text{if } f < 0 \end{cases}$$

Format (ABS)

ABS(f)

where f is a numeric formula.

$$\text{ABS}(f) = \begin{cases} -f & \text{if } f < 0 \\ f & \text{otherwise} \end{cases}$$

Format (LIM)

LIM ($f_1, \{f_2\} \{, f_3\}$)

where the f 's are numeric formulas and $f_2 < f_3$.

$$\text{LIM}(f_1, f_2, f_3) = \begin{cases} f_2 & \text{if } f_1 < f_2 \\ f_3 & \text{if } f_1 > f_3 \\ f_1 & \text{otherwise} \end{cases}$$

The LIM function returns f_1 limited to the range of f_2 to f_3 . If f_2 is missing:

$$\text{LIM}(f_1, , f_3) = \begin{cases} f_3 & \text{if } f_1 > f_3 \\ f_1 & \text{otherwise} \end{cases}$$

Library Subprograms

Similarly, if f_3 is missing:

$$\text{LIM}(f_1, f_2) = \begin{cases} f_2 & \text{if } f_1 < f_2 \\ f_1 & \text{otherwise} \end{cases}$$

Format (RND to a Constant Value)

RND (f, c)

f is a numeric formula and c is a constant numeric formula.

RND (f, c) returns f rounded to accuracy c. It is computed as follows: Let bit position i of f correspond to c such that the value of bit i is greater than c and the value of bit i+1 is less than or equal to c. A quantity which is all zeros except for a 1 in bit position i+1 is added to f. Then, all bits of the result to the right of (but not including) bit i are cleared to 0. This value is returned as the value of RND.

Format (RND for Assignment)

RND (f, v)

f is a numeric formula and v is a numeric variable.

The purpose of this function is to round f for assignment to v. RND (f, v) returns f rounded to an accuracy which is the value of the least significant bit of item v.

Format (PACK)

PACK (f, $c_1, c_2=v$)

f is a numeric or logical formula, the c's are integer constants, $c_1 \leq c_2$, and v is a variable.

PACK (f, $c_1, c_2=v$) is a procedure which results in the setting of bits c_1 through c_2 of v to the value of the $c_2 - c_1 + 1$ low-order bits of f. All other bits of v are not altered. Bits c_1 through c_2 of v should be initially zero, as

PACK will not clear them before ORing f into them. The function is equivalent to the following assignment statement:

$$v = (f \text{ LAND } 2^{*(c_2 - c_1 + 1) - 1}) \text{ LSH } (\pi_f - c_2) \text{ LOR } v$$

where π_f is the bit number of the low-order bit of f .

Example

Assuming a 6-bit machine:

```
A = B'001011'
B = B'000010'
PACK (A, 0, 3=B)
```

results in B being set to 101110_2 .

Format (UNPACK)

UNPACK (f, c_1, c_2)

f is a numeric formula and the c 's are integer constants.

UNPACK (f, c_1, c_2) returns bits c_1 through c_2 of f right-adjusted.

Example

Assuming a 24-bit machine:

```
ABLE = O'533001'
BAKER = UNPACK (ABLE, 6, 14)
```

results in BAKER being set to 533_8 by the assignment.

Format (UNPACX)

UNPACX (f, c_1, c_2)

f is a numeric formula and the c 's are integer constants.

| |
|---------------------|
| Library Subprograms |
|---------------------|

UNPACX (f, c_1, c_2) is the same as UNPACK (f, c_1, c_2) except that sign extension occurs. That is, if bit c_1 is a 1, the value returned will be negative. In the example above,

BAKER = UNPACX (ABLE, 6, 14)

results in BAKER being set to 77777533_8 .

DIRECTIVES

Directives afford the programmer control over the operation of the compiler. CLASP directives are provided for use in debugging a program and determining its execution time, for optimizing execution time and storage requirements, and for inserting direct machine code or assembly language into a CLASP program.

Debugging Directives

The debugging directives TRACE and UNTRACE are used to cause the printing of the names and values of indicated variables and the statement labels encountered during execution of the code they surround. They may appear anywhere in a grouping of executable statements. They do not cause the generation of object code; rather, they generate information to be used by a computer simulator in producing appropriate diagnostic messages when simulation of the object program's execution is performed.

Format

TRACE {variable{, variable}₀[∞]}

{statement}₁[∞]

UNTRACE

Examples

- (1) TRACE TEST, FIND
statement
statement
statement
UNTRACE

The compiler will output information such that when object program execution is simulated, the values and names of variables TEST and FIND will be printed out when they are assigned, along with all statement labels encountered.

- (2) TRACE
statement
statement
UNTRACE

The compiler will output information for a simulator such that a trace would be initiated and would print out statement labels for the code enclosed by the TRACE and UNTRACE commands.

The timing directives COUNT and UNCOUNT are used to time the execution of any object program block they surround. They generate information for use during simulation of the object program's execution and have no effect on the generation of object code itself. The initiating directive COUNT turns on a specified timer in a computer simulator, and the terminating statement UNCOUNT causes the elapsed "real time" consumed during simulated execution of the indicated program block to be printed out. The timers can be used in an overlapping fashion.

Format

COUNT(n)

{statement}₁[∞]

UNCOUNT(n)

n, an integer constant between 0 and 5, indicates the particular timer to be used for accumulating execution time.

Example

```
COUNT(3)
statement
statement
UNCOUNT(3)
```

The COUNT directive initiates the recording of time required to execute the intervening statements and specifies the use of timer 3. The printout of the elapsed time is made during simulation of the object program's execution after the statement preceding the UNCOUNT directive is executed.

Optimization Directives

Three pairs of optimization directives are provided: OPTIMIZE TIME and UNTIME for designating areas of code to be optimized for object program execution time; OPTIMIZE SPACE and UNSPACE for designating areas to be optimized for object program storage requirements; and SIC and UNSIC for exempting specified statements from optimization.

Format (Time and Space Optimization)

OPTIMIZE TIME(n)

OPTIMIZE SPACE(n)

{statement}₁[∞]

{statement}₁[∞]

UNTIME

UNSPACE

The integer constant n specifies the degree or level of optimization to be attempted. The permitted values for n are compiler dependent.

The value specified for n is significant only by comparison with the values specified in other optimization directives. For example, OPTIMIZE TIME(20) might indicate that time optimization for a certain area of code was much more important than if OPTIMIZE TIME(1) had been used. The two optimization directives can be overlapped. If OPTIMIZE TIME(20) and OPTIMIZE SPACE(1) were applied to the same region, that region would be optimized chiefly with regard to time, with much less attention to space optimization.

Format (Optimization Exemption)

SIC

{statement}₁[∞]

UNSIC

No optimization will be performed for the statements between the SIC and the UNSIC, even though optimization directives occur between them or surround them.

The direct code directives **DIRECT** and **END** are used to delimit areas of the program written in direct machine code or assembly language rather than in CLASP. Such code is machine dependent and will vary greatly in both content and form; the conventions for the particular hardware for which CLASP is implemented must be followed.

Format

DIRECT

$\{\{\text{label}\}\{\text{hardware-instruction}\}\}_1^\infty$

END

Items declared in the CLASP program may be referenced in the direct code by their identifiers, and transfers may be made from direct code to labeled CLASP statements. Similarly, a CLASP GOTO may reference a direct code label.

Example

DIRECT

A32. L AC1,L32
ST AC1,M52

END

This is an example of direct code written in IBM 360 BAL.

PART III - COMPILER CONSIDERATIONS

Programs written in CLASP (source code) will be translated into the machine language (object code) of a specific target computer by a compiler. This compiler will not execute on the target computer, but on a large-scale general-purpose computer. Time for compilation is much less important than the execution time and size of the resultant machine code; hence compiler writing techniques that are unfeasible for other applications can profitably be used.

Optimization Techniques

Various optimization techniques should be used to increase the efficiency of the object code. Some of these techniques improve an object program in ways that an intelligent programmer could have achieved by writing a different and better source program; others do so by improving the source-to-object translation process in ways that duplicate what a very competent assembly language programmer would do. This latter type of optimization is particularly important for the CLASP compiler.

Many optimization techniques require an increase in memory to save execution time or vice versa; still others may save execution time in one area of the program at the expense of an increase in the execution time of other areas. The CLASP directives OPTIMIZE TIME (n) and OPTIMIZE SPACE (n) should be used by the compiler to determine the appropriate technique for any given area, or will be used by a particular technique to determine the course of action to be followed. For example, a vector operation such as

$$A(*) = B(*) + C(*)$$

can be implemented in two ways. In areas of a program with no optimization control or space optimization, the code generated for the operation would be a loop requiring the loading, incrementing, and testing of an index register. In areas of the program requiring time optimization, repetitive code might be generated (if the array were small enough) to avoid the execution time overhead of loops.

Scaling optimization, the selection of radix points in fixed-point data to minimize shifting due to rescaling in arithmetic operations, is an important technique for fixed-point target computers. When declaring fixed-point items, the CLASP user will not specify the allocation of more than the number of bits actually required for the expected range of values and necessary accuracy. The compiler must use the resultant flexibility to generate a consistent set of scalings that minimizes intermediate shifting of items in arithmetic formulas.

The scalings chosen should satisfy following criteria:

- (1) For those segments of the source program controlled by the time optimization directive, the following should be minimized:

$$Cr_1 = \sum_{i=1}^m C_i \cdot N_i$$

where m is the number of segments, i , which are time-optimized; N_i is the argument of the OPTIMIZE TIME directive in the i th segment; and C_i is the number of memory cycles required to execute all shift instructions used to rescale fixed-point quantities in the i th segment.

- (2) Assuming the scalings determined to satisfy (1) are fixed, for those segments of the source program controlled by the space optimization directive, the following should be minimized:

$$Cr_2 = \sum_{i=1}^k S_i \cdot P_i$$

where k is the number of segments, i , which are space-optimized; P_i is the argument of the OPTIMIZE SPACE directive in the i th segment; and S_i is the total number of shift instructions used to rescale fixed-point quantities in the i th segment.

- (3) Assuming the scalings determined to satisfy (1) and (2) are fixed, for those segments of the source program not using time or space optimization, the following should be minimized:

$$Cr_3 = \sum_{i=1}^l R_i$$

where l is the number of segments of source code, i , in which neither time nor space optimization is present; and R_i is the number of shift instructions used for rescaling fixed-point quantities in the i th segment.

Practically, it may not be possible to exactly satisfy these criteria, but the compiler should have algorithms which pick scalings which reduce intermediate shifting. The scaling optimization phases should scan the arithmetic calculations in the CLASP source program to determine which of them use

fixed-point data having more than one possible scaling. For each such calculation, it should determine allowable scalings which minimize the number of rescaling shift instructions (when space optimization is required) or which minimize the time required to execute rescaling shift instructions (when time optimization is required). The optimizer phases should then assign scalings. Failing to eliminate all scaling readjustments (and this is likely for large programs), the optimizer should give priority to scalings specified in calculations in which a high level of optimization is required, and to scalings which satisfy requirements from a comparatively large number of calculations.

Another important optimization technique exists with regard to the storage allocation of items and constants. For target computers that use an extension or base register for normal instruction addressing because the address field of their instructions is not large enough to reference all storage locations, the compiler should allocate items to memory locations so as to minimize the number of instructions needed to load and maintain such registers. Thus, items may not be allocated to core sequentially as they are declared, but should be strategically located. Heuristics, which to some extent are a function of the target computer memory organization, should be developed to group quantities which are used together in calculations so that they may use the same base address or none at all. Items declared with the attribute `CONSTANT` may be duplicated in several areas in the memory to place them near instructions referencing them. Items declared with the attribute `PARAMETER` can also be duplicated, although this is less desirable. If conflicts arise as to the best location for items, those storage assignments should be given priority which minimize base address manipulations in segments of the program under optimization control and minimize these manipulations for a large number of calculations.

On multiregister target computers such as the IBM 4 Pi, effective register assignment is an important optimization technique. Those variables and constants which are referenced most frequently within a loop should be assigned to registers in such a way as to minimize load and store instructions required. The most obvious example of effective register allocation is to assign the loop variable of a `FOR` statement to an index register because the loop variable is often referenced as an index for arrays. Loops are good candidates for register allocation because a loop is executed several times; thus the saving of a few load and store instructions in a loop effects an execution time savings of these instructions multiplied by the number of iterations through the loop. Additionally, information supplied by the `CLASP` programmer should be used to aid in effective register usage. When the programmer uses the `INDEX` declaration to inform the compiler of integer variables which have a high priority, the compiler should attempt to maintain them in index registers.

The compiler should employ the basic optimization technique of register recollection, keeping track of the quantities which remain in registers at the end of each statement and using this information to decrease the number of load instructions generated. Thus, if a value remains undisturbed in a register from a previous statement, there may be no need to recalculate or reload that value if it is required in the current statement being processed. If a statement label appears between the statement in which the register was loaded and the current statement, the register may not contain the proper value since a transfer may occur to the label from other parts of the program; the correct value must be loaded as a consequence of that label's occurrence.

As mentioned earlier, some optimization techniques accomplish largely the same things that could have been done in writing the source program, and the CLASP compiler, although putting less emphasis on these techniques, must make provision for them. Elimination of common expressions is one such optimization technique that should be implemented. For example, computing the assignment statement

$$X = T + (Y + 3 * Z) + W / (Y + 3 * Z)$$

appears to require four additions, two multiplications, and one division. However, the expression $Y + 3 * Z$ appears twice in the statement. It should be computed once and saved for use in the division operation, with the result that one less multiplication and one less addition would be required.

Other such optimization techniques that should be implemented include reducing the strength of operations, that is, replacing some exponentiations with multiplications and some multiplications with additions or shifts. Fixed-point divisions by a constant should be replaced by a multiply operation if this operation has a shorter execution time. Also, compile time evaluations of operations between constants should be done. The compiler should permute operands of multiplication and addition operations in order to combine constants at compile time. For example, the assignment statement

$$X = 3.0 * Y / 4.0$$

should be changed into the equivalent statement

$$X = 0.75 * Y$$

without the divide operation.

Compiler Outputs

The most important output of the compiler will be the object program in a form suitable for the target computer. This output should be in assembly language format in spite of the increase in compilation time that this might require. If an assembler is available for the target computer, the compiler should produce assembly code in a format compatible with it. If an assembler does not exist, one should be written.

The printed output generated by the compiler must be much more extensive than that obtained from conventional compilers, for which the goal is usually to produce simple, machine-independent listings oriented to the least sophisticated user. The CLASP compiler output should be oriented to the experienced programmer and should fully document the program's machine-dependent aspects. In areas in which CLASP gives the programmer control on a close-to-hardware basis, the output listings should tell him if he is accomplishing what he wants correctly. In other areas, the programmer simply declares conditions that the program must meet, and the output listings should indicate the decisions made by the compiler to achieve these requirements. Comprehensive output listings of both compiler final results and intermediate results should be printed. These will be of great benefit in studying the efficacy of the optimization and scaling algorithms.

The printed output listing of the source program should be extensively annotated with information obtained at various compilation phases. One type of annotation is the print of errors detected in the source program. Three categories of errors should be printed: warnings, syntactic errors, and compiler capacity overflow.

Warnings should be printed when it appears to the compiler that an execution error may occur, when potentially marginal arithmetic calculations are performed, and when gross inefficiencies are introduced into the source code. For calculations involving division of intermediate expressions, the compiler should print the numerical range of those intermediate expressions for which the generated object code will be valid.

Syntactic error messages should be printed when the compiler is unable to interpret the meaning of the source code or can only assign an ambiguous meaning to it, for example, when parentheses do not balance; when a misspelled or undefined primitive has been used; or when a nonsubscripted item has been used with a subscript.

Compiler capacity error messages are required because, as for any compiler, there will exist otherwise valid source programs which overextend the

compiler's abilities. Usually these types of errors can be eliminated by dividing the program into segments and compiling them separately or by rewriting the statements causing the overflow. Some of this class of errors are global in nature and will cause immediate termination in compilation, including the error scan of the remainder of the program. One such error is expected to be the case in which a CLASP programmer has defined too many symbols for the compiler to handle. Other compiler capacity errors are local and should discontinue the scan only on a particular statement, so that the error scan will continue for the remainder of the program. For example, when parentheses are nested too deeply to compile, the error should stop the scan in the current statement and cause a skip to the next statement. A table which handles the level of nesting should be reinitialized so that the error scan can continue.

Two cross-reference tables should be produced to aid the programmer in analyzing and modifying his program. The first is a label cross-reference table containing all labels sorted alphabetically, their compiler-assigned addresses, and a list of numbers of statements which refer to the labels. The program listing should have sequential numbers attached to each statement for reference purposes. The second table is a cross-reference listing of all declared items and literal constants sorted alphabetically, their compiler-assigned addresses, and statement numbers indicating where the item or constant was referenced.

A list of fixed-point items should also be printed with the scaling determined for each item, since the compiler may have a choice of scalings. This will aid the programmer in choosing scalings for quantities used in modifications to his program and is also necessary so that parameter values may be correctly scaled prior to loading into the target computer.

Compiler Capacities

The minimum compiler capacities should be as follows:

| <u>Program Parameter</u> | <u>Minimum Capacity</u> |
|---|-------------------------|
| Number of names, including data item names, statement names, and procedure, function, and close names | 6,000 |
| Number of characters of source code | 320,000 |
| Number of levels of nested parentheses | 15 |
| Length of object program in words | 16,384 |

Implementation-Dependent Language Features

CLASP is described in Part II independently of any target computer or implementation. When a compiler is developed for a specific computer, certain language characteristics and features will have to be clarified or altered in light of the implementation. Following is a list of language characteristics to be defined:

- (1) Allowable characters that may be used in text or comments
- (2) Maximum precision obtainable with fixed-point numeric quantities
- (3) Range of values for integer quantities
- (4) Hardware codes for machine registers and interrupts
- (5) Format of direct code

APPENDIX A - INDEX FOR PART II

| | | | |
|--|----------|---|-------|
| A (fixed-point type declarator) | 31, 34 | CONSTANT (data item attribute | 31 |
| ABS (absolute value function) . | 115 | in Boolean declarations | 39 |
| Addition operator (+)..... | 59 | in fixed-point declarations... | 34 |
| AND (Boolean operator)..... | 67 | in floating-point declarations | 37 |
| Arrays: | | in integer declarations | 38 |
| fixed-dimension..... | 41 | in textual declarations | 40 |
| variable-dimension | 44 | Constants: | |
| Assignment | 71 | binary | 28 |
| exchange | 85 | Boolean | 26 |
| fixed-point, using TEMP.... | 75 | fixed-point..... | 22 |
| multiple..... | 79 | floating-point | 24 |
| nonscalar..... | 81 | hexadecimal | 28 |
| type conversion in..... | 72 | integer | 25 |
| B (binary constant indicator).. | 28 | location | 30 |
| B (Boolean type declarator)... | 31, 39 | octal | 28 |
| Binary constants..... | 28 | textual | 27 |
| BOOLEAN (Boolean type declarator)..... | 31, 39 | COUNT (timing directive start delimiter..... | 121 |
| Boolean: | | Data declarations: | |
| constants | 26 | array, fixed-dimension | 41 |
| data declarations | 39 | array, variable-dimension .. | 44 |
| formulas | 66 | Boolean | 39 |
| operators, table of | 67 | fixed-point .. | 34 |
| BY (loop statement step size indicator) | 95 | floating-point | 37 |
| Character set | 14 | group | 46 |
| Chronic statements | 100 | hardware | 48 |
| CLOSE (close declarator)..... | 111 | implicit subscript | 45 |
| Closes | 111, 113 | index | 47 |
| Comments | 15 | integer..... | 38 |
| Conditional statements | 92 | textual | 40 |
| | | DECLARE (data item declarator)..... | 31-49 |
| | | Debugging directives | 120 |
| | | DIRECT (direct code start delimiter)..... | 123 |
| | | Direct code directives | 123 |

| | | | |
|----------------------------------|---------------|-----------------------------------|--------|
| Directives: | | Floating-point: | |
| debugging | 120 | constants | 24 |
| direct code | 123 | data declarations | 37 |
| optimization | 122 | FOR (loop statement | |
| timing | 121 | start delimiter) | 95 |
| Division operator (/) | 59 | Formulas: | |
| ELSE (ELSE statement group | | Boolean | 66 |
| start delimiter) | 92 | logical | 62 |
| Enable statements | 98 | numeric | 59 |
| END (conditional and loop | | textual | 53 |
| statement group delimiter) | 92, 95 | Functions | 110 |
| END (direct code delimiter) .. | 123 | GOTO statements | 89, 90 |
| ENDALL (conditional and | | GQ (relational operator) | 66 |
| loop statement group | | GR (relational operator) | 66 |
| delimiter) | 92, 95 | Group declarations | 46 |
| EQ (relational operator) | 66 | HARDWARE (hardware | |
| EQUIV (Boolean operator) ... | 67 | register operand declarator) .. | 48 |
| Exchange assignment | 85 | Hardware: | |
| Exchange operator (==) | 85 | declarations | 48 |
| EXIT (chronic statement | | interrupts | 100 |
| delimiter) | 100 | Hexadecimal constants | 28 |
| EXIT (procedure, function, | | I (integer type declarator) | 31, 38 |
| and close delimiter) .. | 105, 110, 111 | Identifiers | 15 |
| Exponentiation operator (**) . | 59 | IF (conditional statement | |
| F (floating-point type | | start delimiter) | 92 |
| declarator) | 31, 37 | Implicit subscript declarations | 45 |
| FALSE (Boolean constant) ... | 26 | Implicitly subscripted items .. | 58 |
| FIXED (fixed-point type | | INDEX (index register | |
| declarator) | 31, 34 | assignment declarator) | 47 |
| Fixed-point: | | Index declarations | 47 |
| assignment using TEMP | 75 | Inhibit/enable statements | 48, 98 |
| constants | 22 | INLINE (procedure attribute) .. | 104 |
| data declarations | 34 | | |
| FLOATING (floating-point | | | |
| type declarator | 31, 37 | | |

| | | | |
|---|--------|--|-----|
| INTEGER (integer type declarator) | 31, 38 | NOT (Boolean operator) | 67 |
| Integer: | | NQ (relational operator) | 66 |
| constants | 25 | Numbering conventions | 19 |
| data declarations | 38 | Numeric: | |
| Interrupts, hardware | 100 | formulas | 59 |
| L (location constant indicator) | 30 | operators, table of | 59 |
| Labels, statement | 88 | O (octal constant indicator) ... | 28 |
| LAND (logical operator) | 62 | Octal constants | 28 |
| Library subprograms | 114 | OFF (Boolean constant) | 26 |
| LIM (limiting function) | 115 | ON (Boolean constant) | 26 |
| Location constants | 30 | ON (chronic statement start delimiter) | 100 |
| LOCK (inhibit directive) | 48, 98 | Operator precedence | 70 |
| Logical: | | Operators: | |
| formulas | 62 | assignment (=) | 71 |
| operators, table of | 62 | Boolean | 67 |
| Loop statements | 95 | exchange (:=) | 85 |
| LOR (logical operator) | 62 | logical | 62 |
| LQ (relational operator) | 66 | matrix multiplication (/*) .. | 81 |
| LS (relational operator) | 66 | numeric | 59 |
| LSH (logical operator) | 63 | precedence of | 70 |
| LXOR (logical operator) | 62 | relational | 66 |
| Matrix multiplication | 81 | scaling (.S) | 77 |
| Metalanguage used in format descriptions | 13 | Optimization directives | 122 |
| Multiple assignment | 79 | OPTIMIZE SPACE (space opti- mization start delimiter) ... | 122 |
| Multiplication operator (*) | 59 | OPTIMIZE TIME (time opti- mization start delimiter) ... | 122 |
| Negation operator (unary -) ... | 59 | OR (Boolean operator) | 67 |
| Nonscalar: | | OVERLAY (storage allocation declarator) | 50 |
| assignment | 81 | Overlay declarations | 50 |
| items | 56 | PACK (packing procedure) | 116 |

| | | | |
|--|----------|---|--------|
| PARAMETER (data item attribute)..... | 31 | Sample program | 19, 20 |
| in Boolean declarations | 39 | Scaling operator (.S)..... | 77 |
| in fixed-point declarations .. | 34 | SIC (optimization exemption start delimiter)..... | 122 |
| in floating-point declarations | 37 | SIGN (sign-determination function) | 115 |
| in integer declarations | 38 | Source code format | 14 |
| in textual declarations | 40 | START (program start declarator)..... | 14 |
| Parameters, procedure | 104, 107 | Statement: | |
| Periods, use of: | | format | 14 |
| with close names .104, 107, 111, 113 | | labels | 88 |
| with function names..... | 110 | STOP statements | 102 |
| with group declarations | 46 | Subprograms | 103 |
| with procedure names..... | 104, 107 | Subscripted items | 54 |
| with statement labels | 30, 88 | Subtraction operator (binary -) | 59 |
| Preset value attribute | 31 | Switched GOTO statements ... | 90 |
| in Boolean declarations | 39 | T (textual type declarator) | 31, 40 |
| in fixed-point declarations .. | 34 | TEMP (data item attribute) ... | 34 |
| in floating-point declarations | 37 | in fixed-point assignment ... | 75 |
| in integer declarations | 38 | with scaling operator | 77 |
| in textual declarations | 40 | TEXT (textual type declarator)..... | 31, 40 |
| Primitives: | | Textual: | |
| definition of | 15 | constants | 27 |
| table of | 16 | data declarations | 40 |
| PROC (procedure and function declarator)..... | 104, 107 | formulas | 53 |
| Procedures | 104, 107 | THEN (THEN statement group start delimiter)..... | 92 |
| Program: | | Timing directives | 121 |
| control | 87 | TO (inhibit/enable statement delimiter)..... | 98 |
| structure | 14 | | |
| Relational operators, table of. | 66 | | |
| REM (remainder function) | 114 | | |
| REMQUO (remainder-and-quotient procedure)..... | 114 | | |
| RND (rounding function) | 116 | | |
| RSH (logical operator) | 63 | | |
| .S (scaling operator) | 77 | | |

| | | | |
|--|--------|---|-----|
| TO (loop statement limit indicator) | 95 | UNSIC (optimization exemption end delimiter) | 122 |
| TRACE (debugging directive start delimiter) | 120 | UNSPACE (space optimization end delimiter) | 122 |
| TRUE (Boolean constant) | 26 | UNTIME (time optimization end delimiter) | 122 |
| UNCOUNT (timing directive end delimiter) | 121 | UNTRACE (debugging directive end delimiter) | 120 |
| UNLOCK (enable directive) ... | 48, 98 | Variable-dimension array declarations | 44 |
| UNPACK (unpacking function) . | 117 | X (hexadecimal constant indicator) | 29 |
| UNPACX (sign-extended unpacking function) | 117 | | |